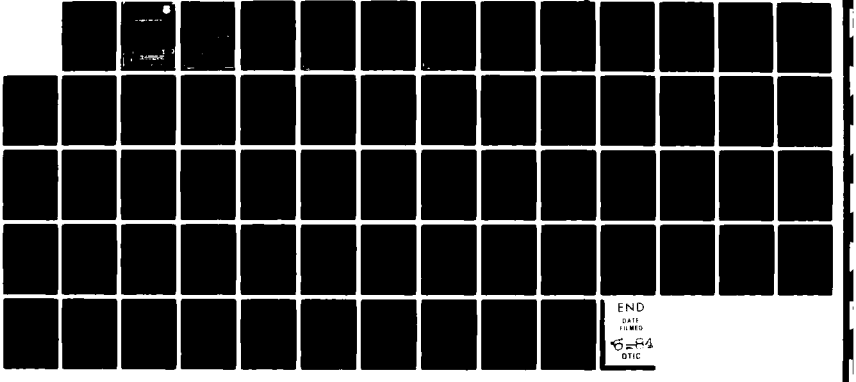
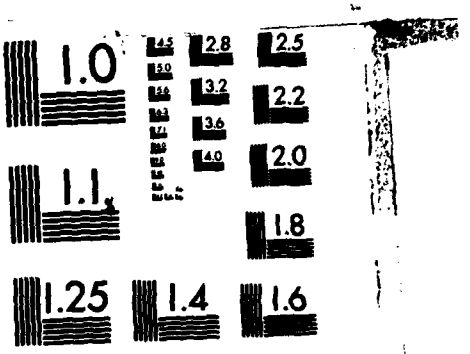
AD-A140 273 ADA ADVANCED ERROR DETECTOR(U) STANFORD UNIV CA 1/1
COMPUTER SYSTEMS LAB D C LUCKHAM JAN 84 RADC-TR-83-299
F30602-80-C-0022

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

RADC-TR-83-377
Final Technical Report
January 1984

ADA ADVANCED ERROR DETECTOR

Stanford University

Dr. David C. Luckham

[redacted] report has been reviewed by the ESD Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be releasable to the general public, including foreign nations.

ESD-TR-83-299 has been reviewed and is approved for publication.

APPROVED: Richard M. Evans

RICHARD M. EVANS
Project Engineer

APPROVED: Raymond P. Utz Jr.

RAYMOND P. UTZ, JR.
Acting Technical Director
Command and Control Division

FOR THE COMMANDER: John A. Ritz

JOHN A. RITZ
Acting Chief, Plans Office

[redacted]

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-83-299	2. GOVT ACCESSION NO. AD-A14027	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ADA ADVANCED ERROR DETECTOR		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report Nov 79 - Nov 82
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) Dr. David C. Luckham		8. CONTRACT OR GRANT NUMBER(s) F30602-80-C-0022
9. PERFORMING ORGANIZATION NAME AND ADDRESS Stanford University Computer Systems Laboratory Stanford CA 94305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62702F 55811907
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COES) Griffiss AFB NY 13441		12. REPORT DATE January 1984
		13. NUMBER OF PAGES 66
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Richard M. Evans (COES)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming Language
Ada Programming Support Environment
Annotation Languages
Deadness Errors
Evasive Action Programming
Exception Propagation
Runtime Error Detection
(See Reverse)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This is the final technical report on a project entitled "Ada Advanced Error Detector." The purpose of this project was to study techniques of detecting common runtime errors in sequential Ada at compile-time using verification techniques, high level annotation languages, and runtime detection of deadness errors in Ada tasking. This work has resulted in a working prototype implementation of a system for detecting and diagnosing tasking errors.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Item 19. Continued:

Runtime Task Monitor
SNOBOL
Source Transformation Preprocessing
Task Scheduling
Task Entries
Rendezvous
Task Deadlock, Blocking, Starvation
Transformation Instrumentation

Table of Contents

Abstract

This is the final report on RADC Project F30602-80-C0022.

Work on Ada advanced error detection has encompassed three areas of research and development.

1. *Techniques of detecting common runtime errors in sequential Ada at compile-time using verification techniques.*

2. *High level annotation languages.*

3. *Runtime detection of deadness errors in Ada tasking.*

Interim Report 1 dealt with work on runtime detection of common errors.

Interim Report 2 contains the project work on a preliminary design study for a high level annotation language for Ada.

This final report deals with our work on runtime detection of errors in Ada Tasking programs.

This report contains a preface summarizing briefly the work in the first two interim reports. The report then deals with the results and progress of our work on tasking error detection. This work has resulted in a working prototype implementation of a system for detecting and diagnosing tasking errors. Source code of this implementation has been supplied to RADC.

This system is the most promising approach to detection of deadness errors and debugging of Ada tasking programs developed so far. Its application to evasive action programming as a standard technique for large Ada distributed systems merits further study. It is our opinion that if this research and development is pursued along the direction taken in this project, runtime monitoring systems of production quality standards for analysis and debugging of Ada tasking programs can be available for incorporation into Ada Program Support Environments in the next 2 - 4 years, depending on the level of effort. Such production quality tools would essentially be developments of the prototype experimental system developed under this contract and described in this report.

Ada Advanced Error Detector

Final Report

November 4, 1979 to November 5, 1982
F30602-80-C-0022

Work on Ada advanced error detection has encompassed three areas of research and development.

1. Techniques of detecting common runtime errors in sequential Ada at compile-time using verification techniques.

2. High level annotation languages.

3. Runtime detection of deadness errors in Ada tasking.

Area 1 was our initial research focus during the first year. This research was based on previous work on the compile-time detection of common errors in Pascal [German 81, Pascal Verifier 79]. This previous work was judged to be a good starting point for the effort since the previous work applied to all Algol-like languages, including the sequential subset of Ada, and therefore promised quick results in terms of application to Ada. Common errors whose presence can be detected at compile-time include accessing of unitialized variables, array indexing errors, sub-range errors, etc. The techniques developed require use of advanced mathematical verification methods such as those implemented in [Pascal Verifier 79]. The advantages resulting are quantified in terms of runtime efficiency of the compiled Ada program gained by suppression of unnecessary runtime checking. The results and details of this part of the research are treated in detail in our Interim Report No. 1 dated 1 February 1981.

The ability to detect errors in the semantics of an Ada program itself, as opposed to a simple common error due to transgressing the general semantics of the Ada language, requires development of a specification language for Ada. Such a language must provide the programmer with sufficiently powerful facilities to express specifications for Ada programs within a syntactic and semantic framework that matches Ada itself where possible. Given both formal specification and Ada text, it is then possible to construct automated interactive debugging tools such as verifiers, test case generators and symbolic executors, for detecting inconsistencies between the specification and code of an Ada program. The major problem lay first in the lack of an adequate specification language for Ada programs. Previous specification languages, e.g. for Pascal, were judged inadequate for application to Ada.

Part of our second year effort was therefore devoted to a design study for a high level specification language called Anna for expressing formal specifications of Ada programs in a machine processable form. In Anna formal comments are written with the same precision as programs, and included as an extension of Ada programs. Formal comments are either virtual Ada text or annotations. Since annotations have a well-defined syntactical structure in ANNotated Ada, they can be processed by tools such as verifiers, optimizers, documentation systems and support tools for program development.

In this preliminary design, we had four principal considerations.

1. Constructing annotations should be easy for the Ada programmer, and should depend as much as possible on notation and concepts of Ada.

2. Anna should possess language features that are widely used in the specification and documentation of programs.

3. Anna should provide a formal framework within which different theories of specifying programs may be applied to Ada.

4. Annotations should be equally well suited for different possible applications, not only for formal verification by also for specification of program parts during program design and development.

The Anna design requirements place heavy emphasis on developing the ways in which Anna can be used for specification and how it may be extended in the future. As a consequence of the choice of a first order annotation language, different theories and techniques of specifying programs may be applied using Anna. For example, previous work on assertional specification of Pascal programs [Hoare 69, Hoare,Wirth 73, Luckham,Karp 79, Pascal Verifier 79] may be formulated in Anna since any programming concept may be defined by the first order axiomatic method (axioms are simply stated as annotations) and used in annotations. It is also clear that the algebraic method of specifying abstract data types may be applied to packages in Anna.

The preliminary Anna design (Interim Report No. 2) is incomplete, and may require further extensions. First, some possibly useful specification concepts are not provided. Consider for instance modal operators. These have to be defined axiomatically at the moment, but it may be useful to include them among the basic predefined operators in later versions. Secondly, Anna does not include tasking. An extension to include task annotations may require the introduction of new predefined attributes, for example task type collections, and the semantics of task annotations will have to be defined.

The Anna specification language is still under development, together with the methodology for compiling formal Anna specifications into Ada runtime checking code. Interim Report No. 2 dated 1 February 1982 gives a preliminary design for Anna developed under this effort, which formed the basis for feasibility studies, and experimentation preliminary to undertaking a more complete design effort.

As pointed out in Interim Report No. 2, the state of basic research concerning errors specific to parallelism in multiprocessing programs has not progressed to a stage where it is practicable to design a formal specification language for Ada tasking. The major portion of our effort related to detection of Ada tasking errors has therefore concentrated on runtime monitoring techniques. Our research and development efforts in this area have been highly successful and promising.

Errors caused by failure in the parallelism of a computational system are called *deadness errors*. These errors are the result of a breakdown in the communication between parallel threads of control in a system. As a consequence, certain threads of control (or sometimes all threads in an entire system) cannot proceed with their computations and hence become "dead". Deadness errors in general occur unpredictably. Whether or not a possible deadness error in a system will occur during system operation may depend on a multitude of external factors, e.g. compilation techniques, runtime scheduling, I/O processing times and external interrupts. They are often extremely difficult to reproduce and hence to locate by current testing methods.

During the second and third years of this research, our major effort has concentrated on development of technology and tools for detection of deadness errors in Ada tasking. Our focus was to provide tools for instrumenting tasking programs so that deadness errors could be detected and diagnostics supplied to the programmer for debugging purposes. This effort has resulted in a significant breakthrough in the area of advanced error detectors for parallel processing.

(i). A transformational method for instrumenting Ada source text in order to monitor for deadness errors in tasking was defined. This method applied to errors due to Ada rendezvous failure. A preliminary paper was published in the 1982 Ada Symposium, [German,Helmbold,Luckham 82].

(ii). An abstract method of representing Ada task rendezvous states and detecting certain kinds of deadness errors was also developed and published in the 1982 Ada Symposium, [German,Helmbold,Luckham 82].

(iii). The transformational instrumentation method was extended significantly to enable detection of deadness errors due to termination failures in Ada tasking. This entailed monitoring of task dependency information (see this final report).

(iv). The transformational instrumentation method was extended significantly to enable monitoring of information for diagnostic description of deadness errors sufficient to enable the programmer to locate the source of an error in the Ada text (see this final report).

(v). The more developed transformational instrumentation was implemented as a Snobol preprocessor for Ada source text (see this final report, Chapter 4).

(vi). A method of monitoring tasking information supplied by instrumented Ada programs (i.e., programs to which the Snobol preprocessor has been applied) and detecting errors and supplying diagnostics was developed and implemented in Ada. This Ada program, consisting of a package and a task is referred to as the *runtime monitor* (described in this final report, Chapter 3).

(vii). Testing of our runtime monitoring system (preprocessor and monitor) on Ada programs led to an experimental development of it for application to *evasive action programming* (see this report).

(viii). The monitoring system was demonstrated at Stanford University to RADC representatives in 13th and 14th July 1982.

(ix). Snobol source text of the preprocessor and Ada source text of the runtime monitor were supplied via ARPA net to the RADC project monitor on December 1982.

This final report contains a theory of deadness errors upon which our runtime detection methods are based, an overview of our prototype runtime monitor and preprocessor designs, and examples of experiments. This system is the most promising approach to detection of deadness errors and debugging of Ada tasking programs developed so far. Its application to evasive action programming as a standard technique for large Ada distributed systems merits further study. It is our opinion that if this research and development is pursued along the direction taken in this project, runtime monitoring systems of production quality standards for analysis and debugging of Ada tasking programs can be available for incorporation into Ada Program Support Environments in the next 2 - 4

years, depending on the level of effort. Such production quality tools would essentially be developments of the prototype experimental system developed under this contract and described in this report.

Professor David C. Luckham
Principle Investigator

1. INTRODUCTION

Errors caused by failure in the parallelism of a computational system are called *deadness errors*. These errors are the result of a breakdown in the communication between parallel threads of control in a system. As a consequence, certain threads of control (or sometimes all threads in an entire system) cannot proceed with their computations and hence become "dead". Deadness errors in general occur unpredictably. Whether or not a possible deadness error in a system will occur during system operation may depend on a multitude of external factors, e.g. compilation techniques, runtime scheduling, I/O processing times and external interrupts. They are often extremely difficult to reproduce and hence to locate by current testing methods.

Deadness errors have been described in the past by concepts such as *deadlock*, *blocking*, and *starvation*. These early concepts provided meaningful classification of certain kinds of errors that could occur in 1960's vintage parallel (or pseudo parallel) systems such as simple operating systems. However they are too vague for describing the kinds of deadness error that can occur in a parallel system implemented using the multi-tasking facilities of Ada. For example, if a system uses dynamic activation of tasks, the number of active tasks at any time will be a function of what the system is doing, and may not be determinable in advance. Names can only be assigned dynamically to new tasks. In such cases, a runtime diagnostic such as "tasks 25, 37, and 121 have deadlocked" will not be very helpful because the dynamically assigned names, 25, 37, 121 have no meaning related to the system source text. Additional descriptive information such as the Ada types of tasks must be provided. Before we can expect to develop an ability to deal with deadness in future parallel systems, we must first provide adequate methods of classification and description.

In order to deal with deadness in Ada or other languages of similar complexity, it is useful to divide the problem into three sub-problems: *(i) detection*, *(ii) description*, and *(iii) avoidance*. Detection involves recognizing a dead state, and usually requires less information than description. Description involves providing sufficient information to locate the source of an error in Ada text. Avoidance involves both style guidelines for constructing error-free systems, and programming techniques for evasion of imminent errors at runtime.

In this paper we investigate the application of runtime monitoring methods to these three sub-problems. Alternative methods of eliminating deadness errors based on static analysis at compile time are not addressed in this paper. So far, the known static analysis methods are very difficult and time-consuming [Taylor 82].

In Chapter 2 a set of concepts for classifying deadness errors in Ada tasking is defined. These concepts are derived from the informal semantics of Ada tasking given in [Ichbiah et al. 82]. They form a complete set in the sense that an operational description of Ada tasking can be given using only these concepts. The description of our implementation in subsequent chapters is based on these concepts. However, we feel that our present set of concepts should be treated as tentative. It is possible to define other complete sets of concepts. Alternative concepts with advantages over the present set may emerge as experience in this area accumulates.

Our monitor system has two parts: (1) a separately compiled runtime monitor written in Ada, and (2) a preprocessor that transforms Ada source text so that necessary descriptive data is communicated to the monitor at runtime. The result of applying the pre-processor to any legal Ada program is a modified program which is again a legal Ada program and contains the monitor. This monitor system

is intended to monitor sufficient information about tasking activities at runtime to *(i)* detect a very broad class of deadness errors, and *(ii)* provide descriptive information about a dead state when it is certain that the state will occur, and prior to that state actually occurring. Some of the basic transformations and an abstract monitoring method were previously described in [German,Helmbold,Luckham 82]. This established the essential ideas behind our implementation but dealt only with detection of deadness and applied to a more limited class of errors.

An Ada implementation of the runtime monitoring system is described in Chapter 3. This description encompasses (a) the kind of descriptive data about tasking states that is monitored, (b) representation of the descriptions and processing to detect errors, and (c) structural design of the monitor. The monitored data must be sufficient both for detection of deadness and for providing diagnostics. The actual monitor data structures and procedures must correctly implement representations of scheduling states (as defined in Chapter 2); any monitor procedure must always terminate, preferably as quickly as possible. Structured design of the monitor is an important consideration both for runtime efficiency and to reduce recompilation if the monitor system is altered for a special application.

Chapter 4 describes the preprocessing transformations applied to Ada source text. The description deals with the complete set of transformations that are currently implemented. These transformations ensure that the monitored program will pass sufficient information about intended tasking operations (initiation, rendezvous, termination, etc.) to the monitor to enable it to detect a wide class of deadness errors, including, e.g., deadness due to inability of dependent tasks to terminate. The fine details are complex; our description is therefore presented informally and relies on illustrative examples. The preprocessor is implemented in SNOBOL.

Chapter 5 deals with the correctness of our method. By this we mean that the addition of the monitor *does not introduce new deadness errors*, and that the monitor correctly describes an error when it is certain that the error will occur if the computation continues normally. Discussion of these issues is informal and proofs are outlined. Our intention here is to indicate how a formal proof can be given; a fully formal treatment is beyond the scope of this paper.

The monitoring system may be used not only for recognition of errors but also for *evasive action* programming. Essentially, the monitor "knows" a deadness error is certain to happen (if the computation continues normally) before it occurs. Warnings (e.g. Ada exceptions) may therefore be propagated to the monitored program before the error occurs, thus enabling it to evade the error by taking some abnormal course of action. Such evasion may be temporary in that the error may become imminent again, but the program can continue useful operation for a time. It may then have to evade again, and so on. These evasive action techniques need to be investigated and developed since they represent a very useful method of keeping large multi-tasking programs in operation. Eventually one would hope to be able to determine at compile time that such programs are free of deadness errors, but until the necessary theory of static detection is developed, evasive action may become just as important a way of dealing with deadness errors as testing methods are for most other kinds of errors today. Indeed, if a system has to deal with unreliable elements, as happens in many practical applications, evasive action techniques could become a standard programming practice.

Some techniques for evasive action programming are given in Chapter 6. These are very modest and represent just a beginning. Examples of monitoring experiments for debugging and evasive action are given in Chapter 7.

The current experimental monitor is programmed in Ada and compiled using the Adam compiler at Stanford [Luckham et al.,ADAM 81]. Since Adam does not support all of Ada82, some parts of the monitor implementation have used circuitous techniques. This is especially evident in our implementation of evasive action; warnings are implemented by means of extra parameters of the monitor entries instead of exceptions because Adam does not support exception propagation during task rendezvous.

There is a fundamental philosophical question as to whether such monitoring should be part of the runtime supervisor package or part of the Ada source text. Basically, it is too early in the development of our understanding of deadness errors to take a stand on this issue. Both approaches have advantages. Supervisor monitoring can make use of scheduling information already present in the supervisor and therefore does not duplicate this information at runtime. However, perhaps standard runtime supervisory packages should not be burdened by requirements to produce debugging information at present, especially since we do not yet know what information is adequate in general. Source code monitoring has many advantages such as the ability to tailor detection information and warnings to a particular application program. The main disadvantage of this approach lies in the lack of a fundamental task identifier type in Ada itself, but this is a problem in programming other resource scheduling applications in Ada too [Luckham et al.,ADAM 81].

2. DEFINITIONS

2.1 TASK STATUSES

According to the semantics of tasking [Ichbiah et al. 82] a task may be in any one of the following statuses; a status has information associated with it:

1. **Running:** a task in this status may be run. This is the only status in which a task may run.

2. **Accepting:** a task t is *waiting for an entry call* at an accept statement or at a select statement that does not have an else clause, terminate alternative, or a delay alternative. The set of entries being waited for (i.e., the entry of the accept or those entries corresponding to open accept alternatives of the select) is associated with the accepting status of t.

3. **Select-terminate:** a task t is at a select statement with a terminate alternative; the set of entries corresponding to open accept alternatives and the set of tasks dependent on t are associated with the select-terminate status of t.

4. **Calling:** task t has issued an entry call, s.e, to task s, which is neither conditional nor timed. The task s and the entry e are associated with the calling status of t.

5. **Block-waiting:** task t has reached the end of an inner block or subprogram and is waiting for the tasks dependent on the inner block to terminate; the set of tasks dependent on the block or subprogram is associated with the block-waiting status of t.

6. **Completed:** task t has completed. The set of tasks dependent on t is associated with the completed status of t.

7. **Terminated:** task t is terminated. No additional information is associated with this status.

8. **Select-Dependents-Completed:** task t is at a select statement with an open terminate alternative and all dependent tasks have reached either terminate status or Select-Dependents-Completed status. The set of entries corresponding to open alternatives of the select statement is associated with this status.

Blocked: A task in any of the statuses 2 - 8 is said to be *blocked*.

This set of statuses and associated information is sufficient to describe that part of the Ada semantics of task rendezvous that determines the schedulability of a task. Such a description may be given by means of a status change diagram indicating how the semantics of rendezvous determines the status changes of a task. Some status changes of task t are *direct* in the sense that the action of t itself causes the change; other changes of t are *indirect* in the sense that they are caused by the action of another task.

Direct Status Changes:

```
running → calling
running → accepting
running → select_terminate
running → block_waiting
running → completed
```

Indirect Changes:

```
calling                          → running
running                          → calling
accepting                        → running
select_terminate                 → running
block_wait                       → running
select_terminate                 → select_dependents_completed
select_dependents_completed      → terminated
select_dependents_completed      → running
completed                        → terminated
```

Notes:
A task executing a delay statement is in status running. The indirect status change from accepting to running occurs when the entry call is issued rather than when the rendezvous is initiated. A task changes from status running to calling after having issued a conditional or timed entry call only if the call is accepted (this status change is therefore indirect). A task which executes a select statement will usually change from running to accepting. A task which executes the else part (or delay alternative) of a select statement remains in status running.

2.2 SCHEDULING STATES AND DEADNESS ERRORS

For a given input a program P may have many different possible computations. Each possible computation is the result of a legal Ada scheduling of the runnable tasks. Here, the word "scheduling" is used in a very broad sense to reflect simply the order in which changes of status occur among the individual tasks of P. Different orders may result from different scheduling algorithms for multiplexing tasks on a single CPU, or from differing speeds of CPU's in a multiprocessor system. The details of the underlying scheduling do not concern us in this paper. We are concerned only with observable differences in the sequence of status changes. It should be noted that different schedulings may result in different outputs from the computation, e.g. in the case where P is monitoring its own status changes.

Task Identifiers. Each task that is activated during a computation of program P is assigned a unique name called its *identifier*. It is assumed that a task can access its own identifier and the identifier of any task that is visible to it.

Task-Status Pairs. A *task-status pair* is an ordered pair consisting of a task identifier as first element and a status as second element (notation: <t,s>).

Scheduling States. A *scheduling state* of a program P is a set of task-status pairs such that each task of P is the first element of exactly one pair. If <t,s> is a member of state S, then task t has status s in S.

Execution. An *execution* of P is a sequence of pairs consisting of a task identifier and a sequence of simple statements such that:

1. the task identifier of the first pair identifies the main program;
2. the task identifier of the *n*th pair $\langle t_n, c_n \rangle$ has status running as a result of the execution of the statements in the previous pairs by the named threads of control;
3. As a consequence of the execution of the statements in the previous pairs in the sequence by the named threads of control, t_n may legally execute the simple statements in c_n in that order, and its status does not change until possibly after the last statement of c_n.

Notes:

Executions correspond to computations of P on a single CPU. An execution is an interleaving of the sequences of simple statements executed by the running threads of control; it is convenient to consider end as a simple statement in the definition of execution.

Statements appear in executions in positions corresponding to their normal termination. For example, if task t calls procedure p, then the simple statements executed during p's execution will appear in an execution pair for t *before* the procedure call. Since a change of status from running occurs during execution of a tasking statement, and possibly back to running again at the completion of that statement, tasking statements will appear at the beginning of sequences in pairs.

The subsequence of pairs representing a single task's executions contains simple statements that the task must execute in a (according to Ada semantics) legal order of execution.

An execution can be constructed from an actual computation in the obvious way by writing down the identifier of the running thread of control at any time followed by the simple statements that it executes. Conversely any execution corresponds to an actual computation on a single CPU under some scheduling. Since the semantics of Ada is independent of the number of CPU's, definitions based on this imposed linearization of tasking computations are valid generally for any method of scheduling computations.

The concept of execution described here can be given a formal definition in terms of transition rules similar to the operational semantics for Ada in [Li 82]. We may therefore use the notions "computation" and "execution" interchangeably in the following discussion.

Scheduling. A *scheduling* is an activity which may change the execution sequence associated with a computation of P given a fixed input.

Notes:

This concept of scheduling is very broad. It includes the implementation of the select statement, relative speeds of processors, computations of the runtime host environment, I/O, and any other activity that may change the order in which different threads of control change statuses.

A program P, given a fixed input, may have many different possible computations, each of which is the result of a change of scheduling.

Sequences of Scheduling States A computation of program P has an associated linear sequence of scheduling states. All tasks are activated in running status. Each new state in the sequence results from the previous state by a status change by one task. Simultaneous status changes are ordered arbitrarily; an indirect status change follows the status change of the task causing it.

Deadness Error. A *deadness error* is a scheduling state occurring in a computation of P in which a subset of tasks are in blocked statuses but not terminated, and there can be no subsequent scheduling state in a possible continuation of that computation of P from that state in which the statuses of the subset have changed.

Potential Deadness Error. A program P has a *potential deadness error* if there is an input and a possible computation such that the associated sequence of states contains that error.

Notes:
A blocked state is a scheduling state in which no task has status running, no (indirect) status changes are possible, and not every task has status terminated.
A deadlocked state is one in which a subset of tasks are all in status calling and the calls are to entries of members of the subset.
Deadness errors include global blocking in which all tasks are blocked, circular deadlock, and errors arising when subsets of tasks block. Implementation dependent errors, e.g. failure of an entry call to be serviced due to a particular implementation of arbitrary selection (starvation), are not included.

2.3 MONITORED PROGRAMS

Runtime monitoring for deadness errors involves adding a monitoring task M to a given program P. The text of P is transformed so that tasks have unique identifiers and may identify each other and communicate status changes to M. The resulting program, P', is called a *monitored* program. It is important to establish that the addition of M to P (to form P') does not change the set of potential deadness errors of P.

The next set of definitions are made in order to establish a sense in which two programs P and P' can be said to possess the same potential deadness errors. As a special case we define what is meant by saying that the *same* deadness error occurs in two distinct computations of P. These definitions are complicated by the possible dynamic creation of tasks in Ada and corresponding dynamic allocation of task identifiers.

correspondence: We assume there is a textual correspondence between P and P' such that:

1. every declaration in P corresponds to a declaration in P' of the same kind,
2. every object in P corresponds to an object (or component object) in P' of the same kind,
3. every statement in P corresponds to a statement in P' of the same kind,
4. the correspondence is consistent, i.e., declarations and statements in a program unit U in P correspond to declarations and statements in the corresponding program unit U' in P'.

Notes:
Any object declared in P corresponds to an object declared in P' of the same kind, in particular tasks correspond to tasks. However, not every declaration or statement in P' need have a correspondence in P.

Corresponding Executions. Let E and E' be executions of P and P' respectively. Assume there is a textual correspondence between P and P'. Then E and E' *correspond* if all task-code pairs of E can be placed in a correspondence with task-code pairs in E' according to the following inductive test: Suppose that the first *n* pairs of E correspond to pairs (in the same order) among the first *m* ($m \geq n$) pairs of E', and that there is a one-one correspondence between all the task identifiers that have occurred so far in E and a subset of those in E'. Let the *n*th. and *m*th. pairs be $\langle t_n, c_n \rangle$ in E and $\langle t_{m'}, c_m \rangle$ in E'.

1. if all the statements of c_m are in (1-1) correspondence (under the textual relationship

between P and P') with the statements of c_n in the same order, then t_n and t_m must correspond. If neither yet corresponds to any task, they are placed in correspondence (in E and E'), and the test proceeds to the next pairs in E and E';
2. If no statement in c_m has a textually corresponding statement in P then $\langle t_n, c_n \rangle$ is compared with the next pair in E';
3. If neither of the first two cases holds then the correspondence test fails.

Notes:
If two executions E and E' correspond then the task identifiers in E are in one-one correspondence with a *subset* of the task identifiers in E'. If t in E corresponds with t' in E' then t executes code corresponding to some of the code executed by t', possibly interspersed with code in E' which has no correspondence in E. Thus, in a general sense corresponding task identifiers are names for threads of control that execute the same subcomputations (restricted to statements of P). E' may have tasks that do not correspond to any task in E; this is a consequence of the assumption that the textual correspondence between P and P' is "into", i.e., P' may be "bigger" than P.

Equivalent Scheduling States. If E and E' are corresponding executions of P and P' then scheduling state S of E is equivalent to a scheduling state S' of E' if for every task-status pair $\langle t, s \rangle$ in S the task-status pair $\langle t', s \rangle$ is in S' where t and t' correspond in E and E', and all other tasks of S' are blocked.

Same Potential Errors. P and P' have the same potential deadness errors if for every potential deadness error of P occurring in execution E, there is a corresponding execution E' of P' in which an equivalent deadness error occurs, and conversely.

Notes:
"Conversely" means the following: if a deadness error S' occurs in execution E' of P' then there is an execution E of P such that E and E' correspond and a deadness error S equivalent to S' occurs in E.

Non Interference. A task M is said not to interfere with a program P if:

1. its addition to P forms a legal Ada program P' and defines a textual correspondence between P and P',
2. P and P' have the same set of potential deadness errors.

Notes:
M does not interfere with P if and only if its (legal) addition to P does not introduce any new potential deadness errors nor remove any potential deadness errors.
The definition of non-interference is weak in the sense that P and P' are not required to compute the same values or to be equivalent in any of the usual senses. The terminology "addition to P" is left undefined; it may involve changes to the text of P as well as the addition of the text of M; it is required that the "addition" sets up a textual correspondence between P and P'.

3. DEADNESS MONITOR.

The monitor detects deadness errors based on information received from the preprocessed program. In our implementation this information consists of changes of statuses and associated information (see Section 2). The monitor maintains, throughout the execution of the modified program, a "picture" of the program's scheduling state. This picture is generally updated and checked for deadness errors whenever information is received from the program. In addition to detecting deadness errors, the monitor also provides facilities for tracing status changes, querying the current "picture" and undertaking evasive action to avoid a deadness error.

3.1 THE MONITOR'S STRUCTURE

Our monitor is implemented in two parts, a task and a package. The task is inserted into the program by the preprocessor. The package is designed to be compiled separately; it contains the monitor's data structure and the procedures that act upon it. This organization allows separate compilation as well as protecting the monitor from simultaneous access.

The package is separated from the program for efficiency reasons. It is compiled only once, and then linked each time a program requires it. Even if several programs are using the monitor, only one copy of the monitor needs to be kept on disk. Separately compiling the monitor also eases the burden on the compiler, the ADAM compiler had troubles dealing with the monitor and a moderate sized program at the same time.

The monitor task's main purpose is to transmit data to the monitor package. The preprocessed program communicates the status change information to the task via the Ada rendezvous mechanism. The task then calls the appropriate procedure of the monitor package. Buffering the information through a task in this way ensures that only one thread of control (the monitor task) can update the monitor's data structure at a time. The monitor task also seems to provide a convenient place to customize the monitor for a specific application, since the monitor's internal workings are hidden in the package. We have created an interactive version of the monitor in this manner.

Structure Outline:

```
package MONITOR_DATA_PACKAGE is
                                        --  Data structures omitted
    procedure ACCEPTING (SERVER         : in TASK_ID;
                         ENTRY_NAME     : in STRING;
                         DEADLK_FLAG    : out BOOLEAN);

    . . .

procedure UNBLOCK (SUBJECT : in TASK_ID);
end MONITOR_DATA_PACKAGE;

task body MONITOR is
begin
    MONITOR_DATA_PACKAGE.INIT;      --  Initialize the monitor package.
    while not MONITOR_DATA_PACKAGE.DONE loop
                                        --  Loop until all other tasks have terminated.
```

```
            select
                                        --  Simply accept a call from the program and
                                        --  relay the information to the monitor.

                accept ACCEPTING (SERVER      : in TASK_ID;
                                  ENTRY_NAME  : in string;
                                  DEADLK_FLAG
                    MONITOR_DATA_PACKAGE.ACCEPTING (SERVER,
                                                    ENTRY_NAME,
                                                    DEADLK_FLAG);
                end;
            or
                . . .
            or
                accept UNBLOCK (SUBJECT : in TASK_ID) do
                    MONITOR_DATA_PACKAGE.UNBLOCK (SUBJECT);
                end;
            end select;
        end loop;
    end MONITOR;
```

3.2 THE MONITOR'S PICTURE

The monitor maintains, at runtime, a picture of the program's scheduling state. This picture consists of status and associated information for each task, entry point information, dependency information, and several global (to the monitor package) counters. This picture is incomplete in that it does not reflect any interactions with the monitor task. All calls to the monitor task are assumed to be promptly answered and completed. The picture may not even be strictly accurate as calls to the monitor may be serviced in an order other than that in which the status changes occur.

3.2.1 TASK INFORMATION

Each activated task of the original program is represented by a record in the monitor's data structure. This record contains status and other information pertaining to the task.

```
    type TASK_RECORD is
                                        --  Each task will have a record of this type to
                                        --  hold information associated with the task.
        record
            TASK_NAME     : MON_NAME_TYPE;
                                        --  A user-defined output name.
            STATUS        : TASK_STATUS;
                                        --  The status of this task.
            CALLED_TASK   : TASK_ID;
                                        --  The task that this task has issued an
                                        --  entry call to.
            CALLED_ENTRY  : MON_NAME_TYPE;
```

```
                                        -- The entry being called.
    PARENT_TASK   : TASK_ID;
                                        -- The task that this one depends on.
    DEPENDENTS    : ID_PTR;
                                        -- A list of tasks depending on this task.
    NUM_WAIT_FOR  : INTEGER;
                                        -- The number of tasks that need to finish
                                        -- before this one can proceed.
    LIST_PTR      : ENT_PTR;
                                        -- A pointer to the list of entries in this task.
    TRACE         : BOOLEAN;
                                        -- True IFF trace information
                                        -- on this task is to be printed
end record;
```

The first field contains the task name. This string is used only to identify the task to the user, and has no internal meaning. The second field contains the task's status (see Section 2.1). The next two fields contain associated information for status calling; the task and entry called. Following these are fields containing dependency information: a list of dependent tasks that this task is waiting on; the number of those tasks that have not terminated; and this task's parent (see 3.2.3). An additional field holds a pointer to the list of entries associated with the task. The last field contains a flag indicating whether or not the task's status changes should be traced. These records are stored in an array, and indexed by task IDs.

3.2.2 ENTRY INFORMATION

The monitor creates entry records for each entry point as it finds out about them (i.e., just before they are referenced at call, accept and select statements). These records contain the unique string representation for the entry created by the preprocessor, the number of tasks calling the entry and a HERE_FLAG, indicating if the task is currently waiting for (ready to accept) a call to the entry. All of the records for a task's entries are stored in an unordered linked list referenced from the task's record.

3.2.3 DEPENDENCY INFORMATION

Keeping track of dependency information poses special problems for the monitor. According to the Ada semantics, each task directly depends on some master (a block, subprogram, task, etc.). This master is usually the scope where the task is declared, however tasks created by an allocator call depend on the scope where the access type was declared.

We define the *sons* of task t (or main program) to be those tasks which:

1. directly depend on t;
2. directly depend on one of t's inner blocks; or
3. directly depend on a subprogram (or subprogram inner block) elaborated by t.

If task s is the SON of task t, then task t is the *father* of task s. This father-son relationship forms a tree structure.

The preprocessor inserts declarations for a list of directly dependent tasks in each block, task and subprogram of the original program (see Section 4.3). An additional list, containing all of the task's sons, is declared in each task body. When new tasks are created, their IDs are added to the appropriate lists during the ADD_DEPENDENTS monitor call.

When task t is ready to terminate, it passes the list of all its sons to the monitor. The monitor sets the PARENT_TASK field in the task record of each task on the list to t's ID. The monitor stores the number of sons that have not yet finished in the t's NUM_WAIT_FOR field.

By checking to see if task t's NUM_WAIT_FOR field is 0, the monitor can easily see if all the sons of t have finished. When this occurs, task t is terminated, along with all of its sons still at select statements with terminate alternatives. The monitor then checks the FATHER field for task t. If it is non-empty (contains a valid task ID) then the PARENT_TASK's NUM_WAIT_FOR count is decremented.

The same algorithm is used when a task attempts to leave a block, except that the list of dependents contains only those tasks that directly depend on the block.

Notes:
It is important to set the PARENT_TASK field of a task only when the father is waiting on that task. Otherwise, the task may decrement the PARENT_TASK's NUM_WAIT_FOR count before the PARENT_TASK is waiting for it (e.g., if the father was waiting on an inner block).
It is important to have the monitor modify these lists of dependents. When a task is attempting to terminate, it passes the monitor a list of it's dependents. If some other task creates a new dependent of the first task, then the change in the list of dependents must be communicated to the monitor. The monitor checks for this situation whenever it updates a dependency list. The monitors mutual exclusion properties are used to ensure that two tasks are never simultaneously updating a dependency list.

3.2.4 GLOBAL VARIABLES

Three variables are used to enable the monitor to efficiently detect global blocking. The monitor maintains counts of:

1. the number of tasks that have been activated;
2. the number that are blocked; and
3. the number that have terminated.

If the number of tasks that are terminated is equal to the number of tasks that have been activated then the program has terminated. Otherwise, if the number of tasks that are blocked is equal to the number of tasks that have been activated, then global blocking has occurred. These checks are done every time a task becomes blocked in the monitor's picture (for any reason).

An additional boolean variable, DONE, is used to inform the monitor task that all of the other tasks have terminated. This variable is declared in the visible part of the monitor package so it can be examined by the monitor task.

Below is the visible part of the monitor package and the specification for the monitor task.

```
-- Data structures used by the monitor.
```

```
                                        --  (I o be compiled separately.)

with DTTY_IO                            --  Adam I/O package.
package MONITOR_DATA_PACKAGE is

                                        --  Bounds and data structures used by
                                        --  the monitor.

    STRING_LENGTH : constant INTEGER := 5;
    MAX_NUM_TASKS : constant INTEGER := 15;
    TASK_LIMIT    : constant INTEGER := (MAX_NUM_TASKS - 1);

    subtype TASK_ID is INTEGER range - 1 .. TASK_LIMIT;
    ALL_TASKS     : constant TASK_ID := - 1;
    NULL_TASK     : constant TASK_ID := - 1;

    subtype MON_NAME_TYPE is string (1 .. STRING_LENGTH);
    NULL_NAME     : constant MON_NAME_TYPE := "#NIL#";

    type ENTRY_REC;
    type ENTRY_PTR is access MON_ENT_REC;
    type ENTRY_REC is
        record
            NAME   : MON_NAME_TYPE;
            NEXT   : ENTRY_PTR;
        end record;
                    --  Used to pass the monitor lists of
                    --  entry points.
    type ID_REC;
    type ID_PTR is access MON_ID_REC;
    type ID_REC is
        record
            ID     : TASK_ID;
            NEXT   : ID_PTR;
                    --  Used to pass the monitor lists of
                    --  task ID's

        end record;                     --  Monitor package procedures are omitted
                                        --  since they correspond one — one with
                                        --  monitor task entries described below.
    . . .
    DONE : BOOLEAN := FALSE;
end MONITOR_DATA_PACKAGE;               --  The DEADLOCK MONITOR task itself.
                                        --  (This is inserted into P.)

use MONITOR_DATA_PACKAGE;

task MONITOR is                         --  Group 1 below are called to notify the
                                        --  monitor of status changes that are about
                                        --  to take place of activation of new tasks,
                                        --  and of task dependencies (see Section 3.1).

    entry NEWTASK (TASK_NAME : in string;
                   NEW_ID    : out TASK_ID);
    entry ADD_DEPENDENT(FATHER : in TASK_ID;
```

```
                        SON     : in TASK_ID);
                        LIST1   : in out ID_PTR;
                        LIST2   : in out ID_PTR;
    entry CALLING (CONSUMER   : in TASK_ID;
                   SERVER     : in TASK_ID;
                   ENTRY_NAME : in STRING;
                   DEADLK_FLAG
    entry ACCEPTING (SERVER     : in TASK_ID;
                     ENTRY_NAME : in STRING];
                     DEADLK_FLAG
    entry SELECTING (SERVER          : in TASK_ID;
                     ENTRY_LIST      : in out ENTRY_PTR;
                     TERMINATE_FLAG : in BOOLEAN;
                     DEPENDENTS      : in ID_PTR;
                     DEADLK_FLAG     : out BOOLEAN);
    entry START_RENDEZVOUS (CONSUMER   : in TASK_ID;
                            SERVER     : in TASK_ID;
                            ENTRY_NAME : in STRING);
    entry END_RENDEZVOUS (CONSUMER : in TASK_ID;
                          SERVER   : in TASK_ID;
                          ENTRY_NAME : in STRING);
    entry END_BLOCK (CONSUMER   : in TASK_ID;
                     DEPENDENTS : in ID_PTR;
                     DEADLK_FLAG
    entry END_TASK (CONSUMER   : in TASK_ID;
                    DEPENDENTS : in ID_PTR;
                    DEADLK_FLAG    : out BOOLEAN);
                                   -- Group 2 provides some facilities for
                                   -- tracing statuses and scheduling states.
    entry PRINT;
    entry TRACE (SUBJECT : in TASK_D;
                 FLAG    : in BOOLEAN);
                                   -- Group 3 is used to facilitate evasive action.
    entry QUERY (SUBJECT    : in TASK_ID;
                 CALLED_TASK, ENTRY_CALLED : out string;
                 WAITING_AT : out ENTRY_PTR);
    entry UNBLOCK (SUBJECT : in TASK_ID);
end MONITOR;
```

3.3 STATUS CHANGES

Calls to the majority of monitor entries are placed in the original program according to the transformation rules given in Section 4. These calls notify the monitor of impending status changes, and any associated information (as defined for each status in Chapter 2). Such calls typically involve modifying the monitor's picture. Below we describe the necessary action that the monitor must take on each call.

Evasive action in this implementation must make use of the DEADLK_FLAG formal parameter of entry calls. All monitor calls which can block the task issuing the call have a BLOCKED_STATE_FLAG actual parameter in addition to those mentioned below. This flag is returned with the value true if and only if a blocked state results in the monitor's picture from the call. For more details on evasive action see Section 6. The DEADLK_FLAG parameter will be ignored for the remainder of this section.

The NEWTASK entry informs the monitor that another task has been created. The monitor creates a new task record, initializing it with the passed task_name and status running. The remaining fields are set to null values. The record is stored in the next available position in the array. The index where it is stored is returned as the NEW_ID.

Notes:
Task ID's cannot be implemented by access type objects accessing task objects because of the strong typing of Ada. The monitor type declarations would have to be changed (and the monitor recompiled) for each monitored program P. The task type declarations of P would have to be placed in the most global declarative part; and still the problem of a task being able to find its own name would remain.

The ADD_DEPENDENT entry is used to put tasks on dependency lists. When the monitor receives this call, it places SON on the two lists. If one of the LISTs is a part of the PARENT's associated information, then the DEPENDENTS list and the NUM_WAIT_FOR count in the PARENT's record are updated accordingly.

The CALLING entry is used to tell the monitor that a task is about to issue an entry call. When the monitor accepts this entry it undertakes the following actions:

1. Change the CONSUMER's status in the monitor's picture from Running to Calling.
2. Increment the queue size (in the monitor's picture) associated with the called entry.
3. If, in the monitor's picture, the SERVER is in status Accepting, Select_Terminate, or Select_Dependents_Completed, and it is waiting on the called entry then the SERVER's status is changed to Running and the NUM_BLOCKED count is decremented.
4. The NUM_BLOCKED count is incremented due to the consumer becoming blocked.

The ACCEPTING entry is used to inform the monitor that a task is about to execute an accept statement. Upon receiving this call the monitor examines the queue-size for this entry. If it is zero, then the SERVERS status is changed to Accepting, the HERE_FLAG for the entry is set, and NUM_BLOCKED is incremented.

SELECTING is called when a task is about to execute a select statement, which may contain terminate alternatives, as well a number of open accept alternatives (see Section 4.4.3). The ENTRY_LIST parameter contains a list of all the entries that can be accepted. The DEPENDENTS parameter holds a list of all the task's sons. The TERMINATE_FLAG parameter will be true only if there is an open terminate alternative. If some of the entries on ENTRY_LIST have non-empty queues (in the monitor's picture), then the SERVER remains in status Running. Otherwise, the HERE_FLAGs for all the entries on the list are set and the TERMINATE_FLAG is checked. If it is true, then

1. The SERVER is placed in status Select_With_Terminate.
2. The SERVER's DEPENDENTS field is set to the passed DEPENDENTS list.
3. If the SERVER's PARENT_TASK field contains a valid ID, then the PARENT_TASK's NUM_WAIT_FOR count is decremented and checked for 0.

If the TERMINATE_FLAG is false, then the SERVER is put into status Accepting. If the SERVER is now blocked, then NUM_BLOCKED is incremented.

The START_RENDEZVOUS entry is called at the start of all the original accept bodies of P. Upon receiving this call the monitor does the following:

1. If the CONSUMER is not in status Calling (e.g. because it issued a conditional or timed entry call) then the actions for entry CALLING are taken. This may cause the SERVER to change status from Accepting to Running.
2. The queue size associated with the entry point is decremented.
3. All of the HERE_FLAGS for the SERVER's entries are cleared, as the server is no longer waiting at any entry.

When receiving the END_RENDEZVOUS entry the monitor simply changes the status of CONSUMER back to Running and decrements the NUM_BLOCKED counter. The server parameter is included for tracing purposes.

The END_BLOCK entry has parameters CONSUMER (the task leaving the block) and DEPENDENTS, a list of tasks which are dependent on the scope being left. If some of the DEPENDENTS have not terminated, the monitor

1. Sets the FATHER field for each task on the DEPENDENTS list to the CONSUMER.
2. Sets the CONSUMER's NUM_WAIT_FOR field to the number of tasks on the DEPENDENTS list that have not finished.
3. Sets the CONSUMER's status to Block_Wait.
4. Increments the NUM_BLOCKED counter.

The END_TASK entry is similar to the END_BLOCK entry, except the CONSUMER is placed in status Completed rather than Block_Wait.

3.4 DEBUGGING/TRACE ENTRIES

These entries are used to control diagnostic output for the monitor, and are placed by the programmer in either the original or transformed Ada source code.

PRINT has no parameters. When the monitor accepts this entry, it prints out its internal picture. Using this, a programmer can get "snapshots" of scheduling states during a computation.

A call to the monitor entry TRACE enables (if FLAG is true) or disables (if FLAG is false) trace output for the SUBJECT. When the monitor receives an entry call whose CONSUMER or SERVER parameter is a task with tracing enabled, then the monitor will display the call and its parameters. It is possible to trace all calls to the monitor by using entry TRACE with parameters ALL_TASKS and TRUE. Normal tracing is restored by calling TRACE with ALL_TASKS and FALSE.

3.5 EVASIVE ACTION

A deadness error is imminent whenever the DEADLK_FLAG parameter has the value true on completion of a monitor call. Evasive action based on testing this parameter value may be programmed in the original source code (see Chapter 6). The two entries UNBLOCK and QUERY are provided to assist this.

UNBLOCK has a single TASK_ID parameter, SUBJECT. The monitor assumes that the SUBJECT task

will *not* proceed with the originally intended rendezvous, and updates its picture accordingly, thus "unblocking" the task. UNBLOCK can be severely misused. It should only be called from the task SUBJECT when the DEADLK_FLAG parameter has been returned true, and SUBJECT is not going to proceed with the tasking statement that has just been indicated by a monitor call.

The entry QUERY may be used to help control evasive action routines. A task passes the monitor the SUBJECT, a TASK_ID, and receives information about how that task is blocked. Specifically, the task and entry that the SUBJECT is calling (if any) and the entries that the SUBJECT is accepting (if any) are returned. This entry is intended to allow more intelligent evasive action by giving the task undertaking the evasive action more information about the error.

4. PREPROCESSOR

This section describes the preprocessor applied to the Ada text of a program P. The purpose of the preprocessor is to introduce communication between the tasks of P and the monitor so that the monitor is informed of any task status change in P. The resulting monitored program is denoted by P'.

The preprocessor applies a sequence of textual transformations. Each transformation introduces new declarations or statements. The transformations currently implemented in our present monitor extend the set of transformations previously given in [German,Helmbold,Luckham 82] in two ways: (1) the set of deadness errors detected by the monitor is extended to include errors involving the inability to terminate, (2) the monitored data is extended to include data necessary to give an adequate description of a deadness error for the purpose of debugging and evasive action. Also the original presentation lacked discussion of many important implementation details upon which the correctness of an actual implementation depends.

The transformations can be broken down into atomic steps describable in a formalism similar to the presentation in [German,Helmbold,Luckham 82]. However formal description of many details (e.g., transformations for composite data structures containing tasks, and for parameter expressions invoking tasking) is very complex. So here our descriptive approach is informal. We describe the preprocessor as a sequence of four passes. First the monitor declaration and body is placed at the beginning of the declarative part of the main program. Following this, each succeeding pass is then assumed to take its input from the output of the preceding pass. Each section of this chapter describes a pass (4.1 - first pass, 4.2 - second pass, etc.). We will use P*k* to designate the output from the *k*th pass, thus P2 is the output from the transformations described in Section 4.2.

The transformations set up a *correspondence* (Section 2.3) between P and P' which is also described informally below.

Notes:
Only the original rendezvous attempts between tasks in P are monitored; rendezvous with the monitor itself are not monitored. All identifiers introduced by the preprocessor, e.g. type names and variables, are assumed not to clash with the identifiers in P.

4.1 INTRODUCTION OF TASK ID'S

Passes 1 and 2 introduce task IDs into the monitored program. Pass 1 introduces data structure to store IDs and communications of IDs; pass 2 introduces code to initialize IDs. The resulting program has the following properties: (1) every active task has a unique ID, (2) a calling task can always access the called task's ID, (3) a task can access its own ID, (4) within every scope the ID of the currently executing task can be accessed, and (5) whenever an entry is called the ID of the caller is passed to the called task.

Notes:
The introduction of task IDs must be done carefully with regard to the Ada semantics of task activation to avoid errors due to accessing uninitialized IDs.

Pass 1 performs the following six transformations:

1. Each task type declaration, t, is replaced by a new task type called t_TASK followed by a record type with the original name, t. t_TASK is obtained by the following modifications to the original declaration, t: A new entry, SET_ID, is placed in the task type declaration, and a new variable, MY_ID, in the task body; an accept SET_ID is inserted as the first statement in the task body. The new record type, t, has two components, a task (called TASK_OBJ) of type t_TASK and a task ID.

2. Each task declaration, t, in P is replaced by a task type declaration t_TASK, a record type, t_RECORD, and a record of that type with the name, t. The task type t_TASK is obtained from the original task declaration by modifications similar to those stated in step1; t_RECORD has two components as above.

3. Entry calls to any task, t.E say, are replaced by entry calls to the task component of the new record, t.TASK_OBJ.E.

4. A new formal parameter called MY_ID of type TASK_ID is added to every subprogram specification.

5. A new formal parameter called CALLER_ID of type TASK_ID is added to every entry specification.

6. All calls to entries and subprograms are modified appropriately as follows: the TASK_ID parameter of every entry and subprogram call is bound to the value of MY_ID. This is either the value of the local MY_ID variable (if the call is in a task) or the value of the formal TASK_ID parameter, MY_ID (if the call is in a subprogram).

Notes:
The ID of the main program always has the value 0.
As a result of step 1, all task object declarations of a task type in P will become declarations of a record type in P1.
As a result of steps 1 and 2 all task objects occur as components of records which also contain a TASK_ID component. We will call these *task records*. If the original tasks were components of a data structure, the new task records take their place in the structure as a result of using the names of the original tasks as names for the task record types (step 1) or task records (step 2).
Wherever a task was visible in P, now both the task and its ID are visible as components of a task record with the same name.
The SET_ID entry and the local MY_ID variable are used to "inform" a task of its own ID when it is activated, and to store that ID.
The Ada semantics do not specify the order of task activation. Therefore at steps 1 and 2 accept SET_ID is inserted as the first statement of every task body; in pass 2 task ID components of all task records are initialized before any task is informed of its ID by a SET_ID entry call. This "holds up" every task until all ID components are initialized, thus avoiding the possibility that tasks in P' might attempt to access task ID components that are uninitialized.
The purpose of steps 4 - 6 is to ensure that the actual value of the CALLER_ID parameter of any entry call is the ID of the task issuing that call. This in turn requires that a subprogram must be able to access the ID of the task that called it so that if it issues an entry call it can pass this ID to the called task. (Note that a subprogram can be visible to, and thus called by, more than one task.) Hence the TASK_ID parameter must be added to both subprograms and entries.

Correspondence: After pass 1, *correspondences* between text of P and new or modified text of P1 is as follows (text in P that is not affected by the transformations corresponds to the same text in P1):

A task object t in P *corresponds* to the task object component of the record with the same name, t, in P1; i.e., t corresponds to t.TASK_OBJ. A task type t in P *corresponds* to a task type in P1 obtained by modifying the declaration of t at step 1 above (called t_TASK). The old and new subprogram and entry declarations and entry calls *correspond*. The new variables MY_ID, entries SET_ID, calls to SET_ID, and new accept SET_ID statements have no correspondence in P.

4.1.1 EXAMPLES OF PASS 1 TRANSFORMATIONS

1. A task type declaration is transformed into a task type followed by a record type:
Note: T1 *corresponds to* T1_TASK.

ORIGINAL TEXT, P:

```
task type T1 is
    entry E1;
    entry E2 (I : in INTEGER; . . .);
end T1;

task body T1 is
    . . .
begin
    . . .
end T1;
```

TRANSFORMED TEXT, P1:

```
task type T1_TASK is
    entry SET_ID (N : in TASK_ID); entry E1 (CALLER_ID : in TASK_ID);
    entry E2 (CALLER_ID : in TASK_ID; I :in INTEGER; . . .);
end T1_TASK;
task body T1_TASK is
    MY_ID : TASK_ID;
    . . .
begin
    accept SET_ID (N : in TASK_ID) do
        MY_ID := N;
    . . .
end T1_TASK;

type T1 is
    record
        TASK_OBJ : T1_TASK;
        ID       : TASK_ID;
    end record;
```

2. All task object declarations become task record object declarations:
Note: A_TASK *corresponds* to A_TASK.TASK_OBJ.

ORIGINAL TEXT, P:

```
A_TASK : T1;
```

TRANSFORMED TEXT, P1:

```
A_TASK : T1;
```

3. Declarations of a single task are transformed into a task type and record type declaration, followed by a record declaration:
*Note:*T1 *corresponds* to T1.TASK_OBJ. **ORIGINAL TEXT, P:**

```
task T1 is
    entry E1;
    entry E2 (N : in INTEGER; . . .);
end T1;

task body T1 is
    . . .
end T1;
```

TRANSFORMED TEXT, P1:

```
task type T1_TASK is
    entry SET_ID (N          : in TASK_ID);
    entry E1     (CALLER_ID : in TASK_ID);
    entry E2     (CALLER_ID : in TASK_ID;
                  N          :in INTEGER; . . .);
end T1_TASK;

task body T1_TASK is
    MY_ID : TASK_ID;
    . . .
begin
    accept SET_ID (N : . . .) do
        MY_ID := N;
    end SET_ID;
    . . .
end T1_TASK;

type T1_RECORD is
    record
        TASK_OBJ : T1_TASK;
        ID       : TASK_ID;
    end record;

T1 : T1_RECORD;
```

4. Pass 1 transformations modify subprogram and entry declarations and calls:

ORIGINAL TEXT, P:

```
procedure PROC1 is
    . . .
end PROC1;

function F1 (I : in INTEGER)
    return SOME_TYPE is
```

```
    . . .
end F1;

PROC1;

K:= F1 (J);

T1.E2 (N);
```

TRANSFORMED TEXT, P1:

```
procedure PROC1 (MY_ID : in TASK_ID) is
    . . .
end PROC1;

function F1 (MY_ID : in TASK_ID; I : in INTEGER)

return SOME_TYPE is
    . . .
end F1;
PROC1 (MY_ID);
K := F1 (MY_ID, J);
T1.TASK_OBJ.E2 (MY_ID, N);
```

4.2 INITIALIZATION OF TASK ID'S

Pass 2 accepts as input the result of Pass 1 and inserts statements to initialize TASK_ID components and variables. When a task record is declared, the declaring scope must call the monitor to obtain a new ID, initialize the ID field of the task record, and inform the task of its ID. If several tasks are declared in the same declarative part then *all* of the ID record components must be initialized before letting any task proceed, otherwise one of the tasks could access an ID component before it has been initialized.

The Pass 2 transformations for initializing the IDs of statically declared tasks in each declarative part are:

1. For each task record declaration a call to MONITOR.NEWTASK is inserted; the string parameter of this call is bound to the task record name and the TASK_ID parameter is the task record ID component. If the declaration is in the declarative part of a subprogram or block the call is placed in the first statement position of that subprogram body or block; if the declaration is in the declarative part of a task body, the call is placed immediately following the accept SET_ID statement of the task body.

2. Immediately following all the calls to MONITOR.NEWTASK inserted at step 1, calls to the SET_ID entry of the task component of each task record are inserted; the TASK_ID parameter of each call is bound to the ID component of the same task record.

If tasks are declared as part of a complex structure (built out of arrays, records, and access types) then Pass 2 uses iterative techniques to construct the initialization code for objects of that complex type. E.g., task IDs occuring as components of arrays are initialized by for loops iterated over the array index type. Details of these techniques are omitted.

Notes:
Calls inserted by step 1 will inform the monitor of the identifier in the source text to be associated with each task (for tracing and debugging) and will initialize all task record ID components. The monitor can then associate its own ID for a task with a name for the task in the source text. If a task occurs as a value in a data structure, the name of the global data structure is used, so in general many IDs may be associated with a source text name. As a result of calls inserted at step 2, all tasks now "know" their ID's, and have been "held up" until *all* ID components are initialized.

Correspondence: Text to initialize task id's added by Pass 2, steps 1 and 2 does not correspond to any text in P.

4.2.1 EXAMPLES OF PASS 2 TRANSFORMATIONS

P1 DECLARATIVE PART:

```
T1 : SOME_TASK_TYPE;
TASK_ARRAY : array (1..5) of SOME_TASK_TYPE;

type TWO_TASKS_TYPE is
    record
        FIRST  : SOME_TASK_TYPE;
        SECOND : SOME_TASK_TYPE;
        N      : INTEGER;
    end record;
TWO_TASKS : TWO_TASKS_TYPE;
```

P2 IMMEDIATELY FOLLOWING BEGIN:

```
                                        --  Text to initialize all task ID components
MONITOR.NEWTASK ("T1", T1.ID);
for I in 1..5 loop
    MONITOR.NEWTASK ("TASK_ARRAY",TASK_ARRAY (I).ID);
end loop;
    MONITOR.NEWTASK ("TWO_TASKS.FIRST", TWO_TASKS.FIRST.ID);
    MONITOR.NEWTASK ("TWO_TASKS.SECOND", TWO_TASKS.SECOND.ID);

                                        --  Text to inform all tasks of their ID's
T1.TASK_OBJ.SET_ID (T1.ID);
for I in (1..5) loop
    TASK_ARRAY (I).TASK_OBJ.SET_ID (TASK_ARRAY (I).ID);
end loop;
TWO_TASKS.FIRST.TASK_OBJ.SET_ID (TWO_TASKS.FIRST.ID);
TWO_TASKS.SECOND.TASK_OBJ.SET_ID (TWO_TASKS.SECOND.ID);
```

Note on Example: Due to Pass 1, SOME_TASK_TYPE is now a task record type.

The situation in which a new task is created and activated by an allocator requires special handling in Pass 2. If P contains an access type accessing a type, T, with task type components, then P1 will contain an access type accessing T which now has task record components. Allocation of an object of type T must not be permitted to make an ID component visible before it is initialized. Our approach is to "hide" such allocators in function calls.

Pass 2 contains a third step:

3. Whenever an access type which designates a type containing task components is declared, Pass 2 inserts a new function declaration to be associated with the access type. This function will take as parameter a value of the access type and return the same value. It initializes all task IDs in the object designated by its parameter. Wherever an allocator is called in P1 to create a new object containing task components, Pass 2 will substitute a call to this new function in P2 with the value of the allocator call as its actual parameter.

Correspondence: The new functions and calls to them have no correspondence in P1. The allocator calls in P1 *correspond* to the allocator call parameters of the new function calls in P2.

Example:

P1:

```
type TWO_TASKS_TYPE is
    record
        FIRST  : SOME_TASK_TYPE;
        SECOND : SOME_TASK_TYPE;
        N      : INTEGER;
    end record;

type TWO_TASKS_REF is
    access TWO_TASKS_TYPE;

TWO_TASKS_PTR : TWO_TASKS_REF;
    .
    .
    .

TWO_TASKS_PTR := new TWO_TASKS;
```

P2:

```
type TWO_TASKS_TYPE is
    record
        FIRST  : SOME_TASK_TYPE;
        SECOND : SOME_TASK_TYPE;
        N      : INTEGER;
    end record;

type TWO_TASKS_REF is
    access TWO_TASKS_TYPE;

function NEW_TWO_TASKS (TEMP : in TWO_TASKS_REF) return
    TWO_TASKS_REF is
                                        -- initialize TASK_IDs in TEMP
begin
    MONITOR.NEWTASK (. . .);
    MONITOR.NEWTASK (. . .);
    TWO_TASKS.REF.FIRST.SET_ID (. . .);
    TWO_TASKS.REF.SECOND.SET_ID (. . .);
```

```
        return TEMP;
    end NEW_TWO_TASKS;

    TWO_TASKS_PTR : TWO_TASKS_REF;
        .
        .
        .
    TWO_TASKS_PTR := NEW_TWO_TASKS (new TWO_TASKS);
```

Notes on Example:
Components FIRST and SECOND were originally task types in P, and have become task record types in P1 as a result of Pass 1. When TWO_TASKS_PTR can be referenced in P2, all of the ID's in the designated object will have been initialized.

4.3 MONITORING OF DEPENDENT TASKS

The monitor cannot detect some dead states without dependency information. For example, a task moves from status BLOCK_WAIT to status RUNNING when all of its dependents declared in the block have terminated. Consequently, a task may be dead as a result of a deadness among its dependents which prevents them from terminating.

In order to deal with such situations, the preprocessor adds a variable designating a list of (dependent) task ids to each block, i.e., each block in P' that corresponds to a block in P contains a list of all tasks dependent on that block. The preprocessor also adds calls to the monitor entry, ADD_DEPENDENTS, with this list as a parameter, whenever a new dependent task is activated. At runtime this list is passed to the monitor by the executing task when a new dependent task is activated. or when the executing task has reached the end of that block. Thus, in this present monitor design, updating of dependents lists and checking for termination is done by the monitor itself.

Pass 3 declares a new local variable, DEPENDENT_IDS, at the beginning of every declarative part of P2, except those in the new subprograms introduced in Pass 2. This variable designates a linked list of all tasks directly dependent on the block where it is declared. An additional variable, ALL_DEPENDENTS, is added to the outermost declarative portion of every task body and the main program. These lists are modified only by calls to the monitor's ADD_DEPENDENT entry. After every NEWTASK call, Pass 3 inserts an ADD_DEPENDENT call to the monitor with parameters: the ID of the task executing the block, the ID of the dependent task, the DEPENDENT_IDS variable, and the ALL_DEPENDENTS variable.

Pass 3 of the preprocessor:

1. Adds the declaration, "DEPENDENT_IDS : MONITOR_DATA_PACKAGE.ID_PTR" at the beginning of each declarative part of P2 except for the new subprograms whose declarations were inserted by pass 2.

2. Adds the declaration "ALL_DEPENDENTS : MONITOR_DATA_PACKAGE.ID_PTR" at the beginning of the outermost declarative part of each task body.

3. Inserts the call MONITOR.ADD_DEPENDENT after each call to the monitor entry, NEW_TASK. The parameters are FATHER => MY_ID, SON => out parameter of preceding NEWTASK call, LIST1 => DEPENDENT_IDS, LIST2 => ALL_DEPDENDENTS.

Notes: Tasks created by an allocator depend on the block where the access type was declared, so their ID's must be added to the DEPENDENT_IDS list corresponding to that block. In P2 these allocator calls are replaced by calls to a new function associated with the access type. This function is declared immediately following the access type by Pass 2. It contains the appropriate NEW_TASK calls. Since pass 3 does not insert a declaration of a local DEPENDENT_IDS, . . . in these function bodies, the immediately global DEPENDENT__IDS variable is visible within these function bodies. These will be the DEPENDENT__IDS variables associated with the declarative parts containing the access type declarations. Therefore the Pass 3 monitor calls to ADD_DEPENDENT placed in the function will have as parameter the DEPENDENT__IDS variable for the block in which the access type is declared.
If a select statement, say, in task T1, has a terminate alternative, then the ID's of all tasks directly dependent on T1, or one of its inner blocks, must be passed to the monitor. The variable ALL_DEPENDENTS designates a list of exactly these ID's.

Correspondence: The text added to P3 in Pass 3 does not correspond to text in P2.

4.4 RENDEZVOUS MONITORING

Pass 4 inserts calls to the monitor entries CALLING, ACCEPTING, SELECTING, START_RENDEZVOUS, END_RENDEZVOUS, END_BLOCK, and END_TASK. These calls inform the monitor of direct and indirect status changes, and associated information arising from rendezvous attempts.

The transformation uses strings derived from the source text identifiers as names of task entries. These names are used to notify the monitor which entry of a task is being called by another task and are crucial in the monitor's internal representation of rendezvous statuses. These entry name strings must name exactly one entry in any given task: no entry can be represented by two different strings, and no string can represent two different entries of the same task. A string could represent several entries, as long as they are all in different tasks. An entry family requires a different string for each member of the family. Finally, the transformation introduces arrays for storing and accessing the names associated with entry families; details of these entry family name arrays are omitted.

4.4.1 THE CALLING ENTRY

Calls to this entry are inserted in a task P immediately before an unconditional, untimed entry call. When a call to CALLING is executed, the monitor will change the status of the task to Calling. As soon as this monitor call finishes and the next statement is executed, the task's actual status will be Calling. Timed and conditional entry calls are not monitored because they do not result in the task changing status (until the call has actually been accepted). The CONSUMER parameter is the ID of the task making the call, i.e., the value of MY_ID. The SERVER parameter is the ID component of the called task's task record. The ENTRY_NAME parameter is the string created by the preprocessor naming the called entry. The DEADLK_FLAG parameter indicates whether evasive action should be taken to avoid a blocked state.

Notes:

Examples:

P3:

```
T1.TASK_OBJ.E1 (MY_ID);
T1.TASKOBJ.ENTRY_FAMILY (EXP) (MY_ID);
T1.TASKOBJ.E2 (MY_ID, PARAMETER);
```

P4:

```
MONITOR.CALLING (MY_ID, T1.ID, "E1", DEADLK_FLAG);
T1.TASK_OBJ.E1 (MY_ID);
MONITOR.CALLING (MY_ID, T1.ID, ENTRY_FAMILY_STR (EXP), DEADLK_FLAG);
T1.TASKOBJ.ENTRY_FAMILY(EXP)(MY_ID);
MONITOR.CALLING (MY_ID,T1.ID,"E2"; DEADLK_FL);
T1.TASKOBJ.E2 (MY_ID, PARAMETER);
```

4.4.2 THE ACCEPTING ENTRY

Pass 4 inserts a call to ACCEPTING immediately before each "simple" accept statement that is not a select alternative (preprocessing of select alternatives is described in 4.4.3). The parameters are: MY_ID (server name), the preprocessor string naming the entry being accepted, and DEADLK_FLAG.

Example:

P3:

```
accept E1 (CALLER_ID) do
    . . .
```

P4:

```
MONITOR.ACCEPTING (MY_ID, "E1", DEADLK_FLAG);
accept E1 (CALLER_ID) do
    . . .
```

4.4.3 THE SELECTING ENTRY

Before executing a select statement a (server) task must inform the monitor of those entries that can be accepted by that select statement. It must therefore evaluate the guards of the select alternatives, including any delay or terminate alternatives. This evaluation must be done once. The resulting values are used both to give the monitor the information associated with the new Accepting status (or Select_terminate status) and to execute the select statement afterwards. Pass 4 inserts declarations of new variables to hold the values of the guards, and text to evaluate the select guards and construct the status information for the monitor.

Pass 4 executes the following text transformations for each select statement in P3:

1. The select statement is enclosed in the body of a new block statement.

2. Boolean variables TEMP1, TEMP2, . . . are declared locally in the new block, one for

each select alternative, and initialized to the guard expression of that alternative, or to TRUE if there is no guard.

3. Boolean variables TEMP_DELAY and TEMP_TERMINATE are declared locally after the previous variables. TEMP_DELAY is initialized to TRUE if there is an else part, to the disjunction of the TEMP variables corresponding to delay alternatives, or to FALSE if there is no else part or delay alternatives. TEMP_TERMINATE is initialized to the TEMP variable corresponding to the terminate alternative if there is one and to FALSE otherwise.

4. A variable ENTRY_LIST of type ENTRY_PTR is declared locally and initialized to null.

5. Ada text to construct the list of entry names corresponding to open accept alternatives is inserted at the beginning of the local block body (i.e., before the select statement). This text is instantiated from a single text template and performs a computation as follows: if TEMP_DELAY is TRUE it does nothing; otherwise it builds a list of entry name strings corresponding to the open accept alternatives and then calls the monitor entry, SELECTING, with parameters: MY__ID, ENTRY__LIST, TEMP__TERMINATE, ALL_DEPENDENTS, DEADLK_FLAG.

6. The boolean conditions in the select alternatives are replaced by the corresponding TEMP variables.

Correspondence: The select statement in P4 corresponds to the original select statement in P3. The new local block, declarations, and new text in P4 has no correspondence in P3, except that calls to functions in the new text corresponds to the original calls in guards in P3.

Notes:

1. If TEMP_DELAY is TRUE the server task cannot enter a blocked state but will remain in status Running.
2. TEMP__TERMINATE is declared even if there is no terminate alternative so that the preprocessor can use a single text template for computing the list of open entries.
3. TEMP_DELAY and TEMP_TERMINATE cannot both be true due to Ada rules for select statements.
4. Construction of the list of entries proceeds as follows: ENTRY_LIST is initialized to null; then for each accept alternative with a true guard condition a new MON_ENT_REC record containing the string representing the entry is allocated. If the entry is part of an entry family, its index expression is evaluated at this point (to correspond with the order of evaluation in the Ada semantics). This record is inserted into the list designated by ENTRY_LIST.

Examples:

P3:

```
select
        accept E1 (CALLER_ID : in TASK_ID) do
            . . .
        end E1;
    or
        accept E2 (CALLER_ID : in TASK_ID);
```

```
                    I          : in INTEGER) do
                . . .
            end E2;
        end select;

        select;
            when FLAG1 =>
                accept E1 (CALLER_ID : in TASK_ID) do
                . . .
            end E1;
        or
            when F (X) => delay 10;
        end select;
P4:
        declare
            TEMP1 : BOOLEAN := TRUE;
            TEMP2 : BOOLEAN := TRUE;
            TEMP_DELAY : BOOLEAN := FALSE;
            TEMP_TERMINATE : BOOLEAN := FALSE;
            ENTRY_LIST : ENTRY_PTR;
            begin
            if not TEMP_DELAY then
                ENTRY_LIST := null;
            if TEMP1 then
                ENTRY_LIST := new ENTRY_REC'(NAME => "E1", NEXT => ENTRY_LIST);
            end if;
            if TEMP2 then
                ENTRY_LIST := new ENTRY_REC'(NAME => "E2", NEXT => ENTRY_LIST);
            end if;
            MONITOR.SELECTING (MY_ID, ENTRY_LIST, TEMP_TERMINATE,
                               ALL_DEPENDENTS, DEADLK_FLAG);
            end if;
                select
                accept E1 (CALLER_ID : in TASK_ID) do
                . . .
                end E1;
            or
                accept E2 (CALLER_ID : in TASK_ID;
                           I         : in INTEGER) do
                . . .
                end E2;
            end select;
        end;
            TEMP1           : BOOLEAN := FLAG1;
            TEMP2           : BOOLEAN := F (X);
            TEMP_DELAY      : BOOLEAN := TEMP2;
            TEMP_TERMINATE  : BOOLEAN := FALSE;
        begin
            if not TEMP_DELAY then
                ENTRY_LIST := null;
                if TEMP1 then
                    ENTRY_LIST := new ENTRY_REC(NAME => "E1",
                                                NEXT => ENTRY_LIST);
                end if;
```

```
            if TEMP2 then
                ENTRY_LIST := new ENTRY_REC(NAME => "E2",
                                            NEXT => ENTRY_LIST);
            end if;
            MONITOR.SELECTING(MY_ID, ENTRY_LIST, TEMP_TERMINATE,
                                                        ALL_DEPENDENTS);
            end if;

            select
                when TEMP1 =>
                    accept E1 (CALLER_ID) do
                    . . .
                    end E1;
            or
                when TEMP2 => delay 10;
            end select;
        end if;
    end;
```

Often text inserted by the preprocessor pass 4 can be omitted. In the first example above, none of the TEMP variables for accept alternatives, nor the corresponding conditional tests on them, are needed. The preprocessor does in fact make some optimizations on the use of the TEMP variables.

4.4.4 ENTRY START_RENDEZVOUS

Pass 4 inserts a call to this monitor entry at the beginning of every accept body, even those inside of select statements. The parameters of the call are: CONSUMER => CALLER_ID (a parameter of the entry call), SERVER => MY_ID, ENTRY_NAME => the name_string associated with the entry being accepted.

Correspondence: This entry call has no corresponding code in P.

4.4.5 ENTRY END_RENDEZVOUS

A call to this entry is placed at the end of every accept body. The parameters of this call are: CALLER_ID, MY_ID, and the string associated with the entry accepted. This entry call does not correspond to any code in P3.

Examples:

P3:

```
accept E1 (CALLER_ID : in TASK_ID);

accept E2 (CALLER_ID : in TASK_ID; I : in INTEGER; . . .) do
    . . .
end E2;
```

P4:

```
accept E1 (CALLER_ID : in TASK_ID) do
    MONITOR.START_RENDEZVOUS (MY_ID, CALLER_ID, "E1");
    MONITOR.END_RENDEZVOUS   (MY_ID, CALLER_ID, "E1");
end E1;

accept E2 (CALLER_ID : in TASK_ID,) (I : in INTEGER; . . .) do
    MONITOR.START_RENDEZVOUS (MY_ID, CALLER_ID, "E2");
        . . .
    MONITOR.END_RENDEZVOUS   (MY_ID, CALLER_ID, "E2");
end E2;
```

4.4.6 ENTRY END_TASK

A call to this entry is inserted at the end of every task body. The parameters are MY_ID (the ID of the task that is completing), ALL_DEPENDENTS (the ID's of all tasks dependent on the completing task), and DEADLK_FLAG. The value returned for DEADLK_FLAG will indicate whether or not the task will cause a blocked state by completing. This entry call does not correspond to any code in P.

4.4.7 ENTRY END_BLOCK

A call to this entry is inserted at the end of each inner block (or subprogram) which has a declarative part. The parameters are the same as for END_TASK, except that the local DEPENDENT_IDS variable takes the place of ALL_DEPENDENTS. Again, this entry call does not correspond to any code in P3.

4.5 FUNCTION CALLS IN TASKING STATEMENTS

The above transformations are inadequate when parameters of tasking statements contain function calls since evaluation of these parameters might also involve tasking.

Example:

P:

```
    . . .
function F1 (ARG: in INTEGER)
    return INTEGER is
T3: SOME_TASK_TYPE;

begin
    . . .
end F1;
    . . .
T1.E2 (F1 (X));
```

P4:

```
    . . .
function F1 (MY_ID: in TASK_ID; ARG : in INTEGER)
    return INTEGER is
```

```
T3: SOME_TASK_TYPE;
    DEPENDENT_IDS : ID_PTR;

begin
    MONITOR.NEW_TASK (. . .);
    MONITOR.ADD_DEPENDENT (. . .);
    MONITOR.SET_ID (. . .);
    . . .
    MONITOR.END_BLOCK (MY_ID, DEPENDENT_IDS, DEADLK_FLAG);
end F1;
. . .

MONITOR.CALLING (MY_ID, T1.ID, "E2");
                                    -- A
T1.TASK_OBJ.E2 (MY_ID, F1 (MY_ID, X));
```

At point A in the above example, T1 is in status Calling in the monitor's picture. However, when the call to F1 is executed, T1 could be put into status Block_Wait waiting for tasks dependent on F1 to terminate. Currently, this will confuse the monitor and may lead it to falsely detect a global blocking situation, or not detect an actual one. The preprocessor therefore moves all function calls out of the tasking statements. This requires additional temporary variables to hold the values of parameter expressions and intermediate values.

Examples:

P4:

```
MONITOR.CALLING (MY_ID, T1.ID, "E2");
T1.TASK_OBJ.E2 (MY_ID, F1 (MY_ID, X));
```

P5:

```
TEMP1 := F1 (MY_ID, X);
MONITOR.CALLING (MY_ID, T1.ID, "E2");
T1.TASK_OBJ.E2 (MY_ID, TEMP1);
```

5. CORRECTNESS OF DEADNESS DETECTION

In this chapter we outline an approach to proving the correctness of our runtime monitoring system. A formal treatment of correctness with detailed proofs is beyond the scope of this paper. We attempt here to indicate what needs to be proved and informal reasoning which can be formalized.

Correctness is taken to mean: *(i)* for any potential deadness error of the original program P there is an equivalent potential deadness error in the monitored program P'; *(ii)* in any computation of P', if the monitor detects a deadness error, it will do so *before* that error occurs and that error will occur if the computation continues normally, *(iii)* certain kinds of deadness errors, including global blocking and circular deadlock will always be detected.

Notes:
(i) means that the monitor does not interfere with the set of potential deadness errors of the monitored program. *(ii)* does not imply that the monitor will detect every deadness error, as defined in Section 2.2, but that any error it can detect in its picture will be a future state of P'. *(iii)* is a completeness result.

5.1 NON INTERFERENCE

The preprocessing (Chapter 4) and addition of the monitor sets up a textual correspondence between P and P'. Declarations of subprograms and entries in P correspond to subprograms and entries with the same name in P'; declarations of task types correspond to task types in P'; task objects of P correspond to task components of task record objects in P'; statements that are untouched by the transformation correspond; calls to subprograms and entries in P correspond to calls to subprograms and entries with the same names in P'; finally the corresponding is consistent (Section 2.3), in particular, the block structure of P is preserved in P'. The monitor, calls to the monitor, and additional object declarations and statements inserted prior to monitor calls do not correspond to text in P.

The following discussion is based on this correspondence between P and P'.

Claim 1. For every execution of P there is a corresponding execution of P' and conversely.

Notes: this depends on the assumptions *(i)* that all tasks of P have the same priority, *(ii)* on properties of the correspondence whereby corresponding statements invoke the same status change (if any) on corresponding tasks, and *(iii)* that all monitor rendezvous terminate. Given an execution E of P, it is then possible to construct a corresponding E' of P' by scheduling the runnable tasks so that corresponding tasks execute corresponding code in the order of E; the monitor M is given super-high priority. We note that every declaration and allocate statement of P corresponds to a declaration or allocate of the same kind, so that at any point in E and E' where correspondence has been established, corresponding tasks have been allocated and activated.

Claim 2. In two corresponding computations of P and P', the sequences of status changes of corresponding tasks, except for monitor rendezvous, are the same.

Notes: This requires noting that not only are the direct status changes of corresponding

tasks the same, but corresponding tasks invoke the same indirect status changes on corresponding tasks in executions that correspond. This latter results from noting that corresponding entry calls are to entries with the same name.

Claim 3. If execution E of P corresponds to execution E' of P' then the only possible task in E' that does not correspond to a task in E is the monitor M.

Notes: This depends on an analysis of the declarations inserted by the preprocessing, showing that any task that could be invoked in P' other than M has code corresponding to code in P in its body.

Claim 4. For each potential deadness error of P, P' has an equivalent potential deadness error, and conversely.

Notes: Let deadness error S occur in execution E of P. Let E' be a corresponding execution of P' (Claim 1). Using Claim 2 and termination of Monitor rendezvous, all tasks of E' that correspond to tasks in E will reach the status of their corresponding task in S. At that point in E', the monitor M must be blocked in accepting status. By Claim 3, the scheduling state of E' is equivalent to S.

The converse can be argued similarly by considering corresponding executions.

Note:
Claim 4 states that adding monitor to P does not interfere with P (see Section 2.3).

5.2 CORRECT PREDICTIVE MONITORING

Consider a monitored program P'. Recall that any execution of P' contains one task, the monitor M, which does not correspond to a task in a corresponding execution of the unmonitored program P. A crucial property of the preprocessing (Chapter 4) is that during an execution of P', any task t that makes a direct change of status in attempting to rendezvous with a task other than M, will call the appropriate entry of M before making that status change, and when t is returned to running status as a result of completion of the monitor call, its next status change normally will be the one signalled. We call this property of the preprocessing *predictive monitoring*. As a result of predictive monitoring, the monitor's representation of the scheduling states of P' is always ahead of the actual scheduling state in any computation.

Claim 5 If at any point in an execution of P' the monitor M has an entry call from task t implying a change of status from running to s, then after the monitor rendezvous with t terminates, t will be in status running and if it continues normally its next status change (if any) is to s.

Notes: This depends on a case analysis of the clauses in the preprocessing. Various complications must be noted, e.g. that any actual parameter expressions of entry calls or select alternatives are "unwound" and their values assigned to temporary variables prior to the monitor call. Note that the claim allows for the case where the scheduler does not run t again in that execution.

Predictive monitoring enables the monitor to update its representation of the scheduling state to

reflect a future state under the assumption that runnable tasks will eventually run. Deadness errors that are represented in the monitor therefore must eventually happen in the execution. The predictive property enables the monitor to signal tasks to take evasive action *before* a deadness error actually occurs.

5.3 MONITOR CORRECTNESS AND COMPLETENESS

Three crucial properties of the monitor implementation must be proved: *(i)* all monitor entry calls terminate; *(ii)* the monitor's representation of scheduling states correctly represents the scheduling state implied by any legal sequence (i.e., a sequence that can occur in a computation) of monitor entry calls from a preprocessed program. *(iii)* the monitor will detect any deadness error that is a consequence of its representation and that these errors include global blocking and circular deadlock. (Certain deadness errors may not be adequately represented, and therefore not detectable.)

Notes:
Arguments supporting these claims must be based on the implementation description given in Chapter 3.
Note that a proof of *(ii)* requires showing that indirect status changes implied by a direct change are represented correctly when an entry call is completed.

6. EVASIVE ACTION

In this chapter we outline some paradigm techniques for programming evasive action in the monitored program. These paradigms use Ada exception propagation, and therefore differ from our present implementation and examples.

Notes:
The facilities provided by our monitor for evasive action depend on procedure parameters (e.g. DEADLK_FLAG parameter of CALLING because of deficiencies in the Adam compiler. similarly, the example experiment given in Section 7.4 does not use exceptions.

It is assumed that the monitor may propagate a number of exceptions signifying the imminent occurrence of different kinds of dead states, for example:

```
GLOBAL_BLOCKING, CIRCULAR_DEADLOCK,
DEPENDENTS_BLOCKED, LOCAL_BLOCKING : exception
```

In most cases the monitor will propagate an exception to the final task whose status change will complete a dead state. We may call this the "offending" task, although it may be no more of an offender than other tasks who have already reached blocked statuses. In more sophisticated monitoring systems, exceptions may be propagated to other tasks in the monitored system.

There are three paradigms for using the propagation of such exceptions in the source text to enable the monitored program to take evasive action.

6.1 MINOR EVASION

The evasive action is taken and then the program proceeds exactly as normal (i.e., as it would have if no deadness exception had been propagated from the monitor). This technique may be used in cases where the imminent error may be avoided, e.g. by freeing a resource and delaying, and then acquiring the resource again.

```
. . .
begin                                    -- Block enclosing monitor call.
    MONITOR.CALLING(MY_ID, S_ID, "E"); -- intention to call S
exception
    when GLOBAL_BLOCKING =>  -- Evasive action
end;
S.E;                                     -- Continue to call S as planned.
. . .
```

6.2 MAJOR EVASION

A major evasion requires the program to disrupt its normal course of action. A standard example would be that the "offending" task reset its local data and return to some previous starting point.

Example:

```
loop
    . . .
    begin                                   --  Monitor call and intended task action
        MONITOR.CALLING (MY_ID, S_ID, "E"); --  are placed in a block;
        S.E;                                --  Normal action is to call S.
    exception
        when GLOBAL_BLOCKING =>             --  Evasive action when ERROR is
    end;                                    --  propagated by the monitor call.
        . . .                               --  Do not call S after evasive action,
                                            --  but continue here.
```

6.3 CATASTROPHE

In a catastrophe there is no hope of "the offending task(s)" continuing to function usefully. If this kind of error is signalled the offender will simply report diagnostics and possibly transmit warnings to other tasks in the program. The reporting can be based on "questioning" the monitor.

Example:

```
task body T is
    . . .
begin
    . . .
    MONITOR.CALLING (MY_ID, S_ID, "E");
    S.E;
    . . .
exception
    when GLOBAL_BLOCKNG =>
--  Report conditions and then die gracefully; do not continue.
end T;
```

7. EXAMPLES

In this chapter we give examples of the pre-processing transformation, of the monitor's output describing deadness errors, and of evasive action.

7.1 A DINING PHILOSOPHERS PROGRAM

The following example is the version of the dining philosophers problem with a potential blocking error given by Hoare in his paper on Communicating Sequential Processes. The example gives the original Ada text and the preprocessed text. The output comments from the monitor describing the blocking error when it occurred are also given.

Blocking can occur as follows.

> All five philosophers can enter the room, sit down at the table and pickup one fork; then all forks will be in accepting status waiting for a PUTDOWN, while all philosophers will be in calling status having called PICKUP for their second fork, and the table will be waiting for either of its entries to be called.

Whether or not this situation will happen depends on the underlying scheduling. The error may never occur or may occur almost immediately, depending on the runtime task supervisor. This is illustrated by the delay statements in the Philosopher task body. If the delay before picking up the second fork is removed, the blocked state will never occur when the program is run with the task supervisor package at Stanford; with this delay, the tasks block before any philosopher eats.

```
with DTTY_IO;
use  DTTY_IO;

procedure ROOM is                          -- The cast of actors: FORKS,
pragma MAIN;                               -- PHILOSOPHERS, and TABLE.

    task type FORK is
        entry PICKUP;
        entry PUTDOWN;
    end FORK;

    task TABLE is
        entry SITDOWN (I : out INTEGER);
        entry GETUP   (I : in INTEGER);
    end TABLE;

    task type PHILOSOPHER;

    type SET_OF_FORKS is array (0 .. 4) of FORK;
    FORKS : SET_OF_FORKS;
                                           -- The scripts: the bodies of the actors.

    task body FORK is
    begin
        loop
            accept PICKUP;
```

```
            accept PUTDOWN;
        end loop;
    end FORK;

    task body TABLE is

        type SEAT_ARRAY is array (0 .. 4) of BOOLEAN;
        SEATS : SEAT_ARRAY := (others => TRUE);
                                        -- True means unoccupied.
    begin
        loop
            select
                accept SITDOWN (I : out INTEGER) do
                    for J in 0..4 loop
                        I := J;
                        exit when SEATS (J);
                    end loop;
                    SEATS (I) := FALSE;
                end;
            or
                accept GETUP (I : in INTEGER) do
                    SEATS (I) := TRUE;
                end;
            end select;
        end loop;

    end TABLE;

    task body PHILOSOPHER is
        SEAT : INTEGER;
    begin
        loop
            delay 1;                    -- Delays are for thought. If a large enough
            TABLE.SITDOWN (SEAT);       -- delay is placed between picking up the
            FORKS (SEAT).PICKUP;        -- two forks then the blocked state occurs;
            delay 2;                    -- If not, the philosophers don't block.
            FORKS ((SEAT + 1) mod 5).PICKUP;
                                        -- This illustrates the dependence of
                                        -- the error on the runtime supervision.
            delay 1;
            FORKS (SEAT).PUTDOWN;
            FORKS ((SEAT + 1) mod 5).PUTDOWN;
            TABLE.GETUP (SEAT);
        end loop;
    end PHILOSOPHER;

    SOCRATES, PLATO, ARISTOTLE, MARX, RUSSELL : PHILOSOPHER;

begin                                   -- The five forks, five philosophers, and the
    null;                               -- table are all activated at this point.
end ROOM;
```

7.2 THE PREPROCESSED AND MONITORED DINING PHILOSOPHERS

The source code of the Dining Philosophers program after pre-processing is given. The reader should compare this version with the original text in Section 7.1 and with the descriptions of pre-processing in Chapter 4.

```
with MONITOR_DATA_PACKAGE; use MONITOR_DATA_PACKAGE;
with DTTY_IO;
use  DTTY_IO;
--  The cast of actors: FORKS, PHILOSOPHERS, and TABLE.
procedure  ROOM  is

                                                    --  The DEADLOCK MONITOR itself.

task MONITOR is
    . . .
read;
task body MONITOR is
    . . .                                           --  Uses MONITOR BASE PACKAGE
end MONITOR;

  pragma MAIN;
--  Variables and new type declarations are inserted by Pass 1, (Section 4.1)
--  to introduce task ids; compare with declarations in Section 7.1.
  MY_ID            : constant TASK_ID := 0;
  MON_DEPEND_ID    : ID_PTR;
  MON_LIST         : ENTRY_PTR;
  MON_DEADLK_FLAG  : BOOLEAN;

  task type MONTYPE_FORK is
      entry SET_ID     (N       : in INTEGER);
      entry PICKUP     (CALL_ID : in INTEGER);
      entry PUTDOWN    (CALL_ID : in INTEGER);
  end MONTYPE_FORK;

  type FORK is
      record
          TSKOBJ : MONTYPE_FORK;
          ID     : INTEGER;
      end record;

  task type MONTYPE_TABLE is
      entry SET_ID    (N       : in INTEGER);
      entry SITDOWN   (CALL_ID : in INTEGER; I : out INTEGER);
      entry GETUP     (CALL_ID : in INTEGER; I : in INTEGER);
  end MONTYPE_TABLE;

  type MONREC_TABLE is
      record
          TSKOBJ : MONTYPE_TABLE;
          ID     : INTEGER;
  end record;

      TABLE : MONREC_TABLE;
```

```
task type MONTYPE_PHILOSOPHER is
    entry SET_ID (N : in INTEGER);
end;

type PHILOSOPHER is
    record
        TSKOBJ : MONTYPE_PHILOSOPHER;
        ID     : INTEGER;
    end record;

type SET_OF_FORKS is array (0 .. 4) of FORK;
FORKS : SET_OF_FORKS;
                                        -- The scripts: the bodies of the actors.

task body MONTYPE_FORK is
    MY_ID             : INTEGER;
    MON_DEPEND_ID     : ID_PTR;
    MON_LIST          : ENTRY_PTR;
    MON_DEADLK_FLAG   : BOOLEAN;
    ALL_DEPENDENTS    : ID_PTR;
begin
    accept SET_ID (N : in INTEGER) do
                                        -- Task waits until its ID is initialized
                                        -- (Section 4.1)
      MY_ID := N;
    end;
loop

      MONITOR.ACCEPTING(MY_ID, "PICKU", MON_DEADLK_FLAG);
      accept PICKUP (CALL_ID : in INTEGER) do
          MONITOR.START_RENDEZVOUS (CALL_ID, MY_ID, "PICKU");
          MONITOR.END_RENDEZVOUS   (CALL_ID, MY_ID, "PICKU");
      end;

      MONITOR.ACCEPTING (MY_ID, "PUTDO", MON_DEADLK_FLAG);
      accept PUTDOWN (CALL_ID : in INTEGER) do
          MONITOR.START_RENDEZVOUS (CALL_ID, MY_ID, "PUTDO");
          MONITOR.END_RENDEZVOUS   (CALL_ID, MY_ID, "PUTDO");
      end;

  end loop;

      MONITOR.END_TASK (MY_ID, MON_DEPEND_ID, MON_DEADLK_FLAG);
  end MONTYPE_FORK;

task body MONTYPE_TABLE is
    MY_ID             : INTEGER;
    MON_DEPEND_ID     : ID_PTR;
    MON_LIST          : ENTRY_PTR;
    MON_DEADLK_FLAG   : BOOLEAN;
    ALL_DEPENDENTS    : ID_PTR;

    type SEAT_ARRAY is array (0 .. 4) of BOOLEAN;
```

```
    SEATS : SEAT_ARRAY := (others => TRUE);
begin
    accept SET_ID (N : in INTEGER) do
        MY_ID := N;
    end;
    loop

        if not FALSE then
            MON_LIST := null;
            if TRUE then
                MON_LIST := new MON_ENT_REC(NAME =>"SITDO",
                                            NEXT => MON_LIST);
            end if;
            if TRUE then
                MON_LIST := new MON_ENT_REC(NAME =>"GETUP",
                                            NEXT => MON_LIST);
            end if;
            MONITOR.SELECTING(MY_ID, MON_LIST, FALSE,
                              MON_DEPEND_ID, MON_DEADLK_FLAG);
        end if;
        select
          accept SITDOWN (CALL_ID : in INTEGER;
                          I       : out INTEGER) do
              MONITOR.START_RENDEZVOUS(CALL_ID, MY_ID, "SITDO");
              for J in 0..4 loop
                  I := J;
                  exit when SEATS(J);

              end loop;
              SEATS(I) := FALSE;
              MONITOR.END_RENDEZVOUS (CALL_ID, MY_ID, "SITDO");
          end;

       or
          accept GETUP (CALL_ID : in INTEGER;
                        I       : in INTEGER) do
              MONITOR.START_RENDEZVOUS (CALL_ID, MY_ID, "GETUP");
              SEATS(I) := TRUE;
              MONITOR.END_RENDEZVOUS   (CALL_ID, MY_ID, "SITDO");
          end;

        end select;

        end loop;

        MONITOR.END_TASK (MY_ID, MON_DEPEND_ID, MON_DEADLK_FLAG);
  end MONTYPE_TABLE;

task body MONTYPE_PHILOSOPHER is
    MY_ID             : INTEGER;
    MON_DEPEND_ID     : ID_PTR;
    MON_LIST          : ENTRY_PTR;
    MON_DEADLK_FLAG   : BOOLEAN;
    ALL_DEPENDENTS    : ID_PTR;
    SEAT              : INTEGER;
```

```
    begin
        accept SET_ID (N : in INTEGER) do
            MY_ID := N;
        end;
        loop
            delay 1;

            MONITOR.CALLING (MY_ID, TABLE.ID, "SITDO", MON_DEADLK_FLAG);
            TABLE.TSKOBJ.SITDOWN (MY_ID,SEAT);

            MONITOR.CALLING (MY_ID, FORKS(SEAT).ID, "PICKU", MON_DEADLK_FLAG);
            FORKS (SEAT).TSKOBJ.PICKUP (MY_ID);
            delay 2;

            MONITOR.CALLING (MY_ID, FORKS ((SEAT + 1) mod 5).ID, "PICKU",
                                                        MON_DEADLK_FLAG);
            FORKS ((SEAT + 1) mod 5).TSKOBJ.PICKUP (MY_ID);
            delay 1;

            MONITOR.CALLING(MY_ID, FORKS (SEAT).ID,"PUTDO",MON_DEADLK_FLAG);
            FORKS (SEAT).TSKOBJ.PUTDOWN (MY_ID);

            MONITOR.CALLING (MY_ID, FORKS ((SEAT+1) mod 5).ID, "PUTDO",
                                                        MON_DEADLK_FLAG);
            FORKS ((SEAT+1) mod 5).TSKOBJ.PUTDOWN (MY_ID);

            MONITOR.CALLING (MY_ID, TABLE.ID, "GETUP", MON_DEADLK_FLAG);
            TABLE.TSKOBJ.GETUP (MY_ID, SEAT);

        end loop;

        MONITOR.END_TASK (MY_ID, MON_DEPEND_ID, MON_DEADLK_FLAG);
    end MONTYPE_PHILOSOPHER;

    SOCRATES   : PHILOSOPHER;
    PLATO      : PHILOSOPHER;
    ARISTOTLE  : PHILOSOPHER;
    MARX       : PHILOSOPHER;
    RUSSELL    : PHILOSOPHER;
begin

--  Monitor calls inserted by Pass 2 (Section 4.2) to initialize all task ids in task records,
--  and track task dependencies (Section 4.3)
    MONITOR.NEWTASK ("TABLE", TABLE.ID);
    MONITOR.ADD_DEPENDENT (MY_ID, TABLE.ID, MON_DEPEND_ID, ALL_DEPENDENTS);
    for MON_I1 in 0 .. 4 loop
        MONITOR.NEWTASK ("FORKS", FORKS(MON_I1).ID);
        MONITOR.ADD_DEPENDENT (MY_ID, MON_DEPEND_ID, FORKS (MON_I1).ID);
    end loop;
    MONITOR.NEWTASK ("SOCRA", SOCRATES.ID);
    MONITOR.ADD_DEPENDENT (MY_ID, MON_DEPEND_ID, SOCRATES.ID);
    MONITOR.NEWTASK ("PLATO", PLATO.ID);
    MONITOR.ADD_DEPENDENT (MY_ID, MON_DEPEND_ID, PLATO.ID);
    MONITOR.NEWTASK ("ARIST", ARISTOTLE.ID);
    MONITOR.ADD_DEPENDENT (MY_ID, MON_DEPEND_ID, ARISTOTLE.ID);
```

```
  MONITOR.NEWTASK ("MARX ", MARX.ID);
  MONITOR.ADD_DEPENDENT (MY_ID, MON_DEPEND_ID, MARX.ID);
  MONITOR.NEWTASK ("RUSSE", RUSSELL.ID);
  MONITOR.ADD_DEPENDENT (MY_ID, MON_DEPEND_ID, RUSSELL.ID);

--  SET_ID calls to inform each task of its ID inserted by Pass 2 (Section 4.2).
  TABLE.TSKOBJ.SET_ID (TABLE.ID);
  for MON_I1 in 0 .. 4 loop
      FORKS (MON_I1).TSKOBJ.SET_ID (FORKS (MON_I1).ID);
  end loop;
  SOCRATES.TSKOBJ.SET_ID (SOCRATES.ID);
  PLATO.TSKOBJ.SET_ID (PLATO.ID);
  ARISTOTLE.TSKOBJ.SET_ID (ARISTOTLE.ID);
  MARX.TSKOBJ.SET_ID (MARX.ID);
  RUSSELL.TSKOBJ.SET_ID (RUSSELL.ID);
  null;

  MONITOR.END_TASK (MY_ID, MON_DEPEND_ID, MON_DEADLK_FLAG);
end ROOM;
```

7.3 DIAGNOSTIC DESCRIPTION OF THE DINING PHILOSOPHER'S DEAD STATE

Below is the description of a global blocking state given by the monitor.

Key: In descriptions of Accepting status, each entry name is followed by it's queue size (an integer) and a "*" if the task is in a status accepting that entry.

```
    **MON**  GLOBAL DEADNESS HAS BEEN DETECTED
    **MON**     TASK INFORMATION

    0 MAIN  is block_waiting on 11 tasks.
        Its entries are:
                <NONE>
      Its father is: -1

--  This description indicates that the table task was in accepting status,
--  accepting either entry, and neither entry had been called.

    1 TABLE is accepting
        Its entries are:
                SITDO (0*)  GETUP (0*)
      Its father is: 0

--  Fork indicated as task 2 is in status accepting PUTDOWN which no callers,
--  while some task has called PICKUP.

    2 FORKS is accepting
        Its entries are:
                PUTDO (0*)  PICKU (1)
      Its father is: 0

    3 FORKS is accepting
        Its entries are:
```

```
            PUTDO (0*)  PICKU (1)
  Its father is: 0

4 FORKS is accepting
    Its entries are:
            PUTDO (0*)  PICKU (1)
  Its father is: 0

5 FORKS is accepting
    Its entries are:
            PUTDO (0*)  PICKU (1)
  Its father is: 0

6 FORKS is accepting
    Its entries are:
            PUTDO (0*)  PICKU (1)
  Its father is: 0

--  SOCRATES is task 7; it has called task 3 (a fork) entry PICKUP;
--  we can see above that task 3 is accepting PUTDOWN.

7 SOCRA is calling task number 3 at entry PICKU
    Its entries are:
            <NONE>
  Its father is: 0

8 PLATO is calling task number 4 at entry PICKU
    Its entries are:
            <NONE>
  Its father is: 0

9 ARIST is calling task number 5 at entry PICKU
    Its entries are:
            <NONE>
  Its father is: 0

10 MARX is calling task number 6 at entry PICKU
    Its entries are:
            <NONE>
  Its father is: 0

11 RUSSE is calling task number 2 at entry PICKU
    Its entries are:
            <NONE>
  Its father is: 0
**MON**  end of dead state description.
```

7.4 THE EVASIVE ACTION PHILOSOPHER TASK

The following is an example of a philosopher task with additional evasive action capability. If the task receives a warning from the monitor upon informing it that the next action is to pickup its right hand fork, the evasive action will be to putdown the lefthand fork. It will then attempt to eat again as before.

This can be programmed using paradigm 6.1, Section 6. It is assumed that this source text will be preprocessed and monitor calls placed as usual, including the evasive action text.

```
task body PHILOSOPHER is
    SEAT : INTEGER;
begin
    loop
        delay 1;
        TABLE.SITDOWN(SEAT);
        FORKS(SEAT).PICKUP;
        delay 1;
        begin
            MONITOR.CALLING(MY_ID,FORKS((SEAT + 1) mod 5).ID "PICKUP");
--  This call might propagate GLOBAL_BLOCKING.
        exception
            when GLOBAL_BLOCKING =>
                FORKS(SEAT).PUTDOWN;
                                        -- Evasive action: put down left hand fork.
                FORKS(SEAT).PICKUP;
                                        -- Try to pick up both forks again.
         end;
         FORKS((SEAT + 1) mod 5).PICKUP;
                                        -- May get same error again here.
         delay 1;
         FORKS(SEAT).PUTDOWN;
         FORKS ((SEAT +1) mod 5).PUTDOWN;
         TABLE.GETUP(SEAT);
    end loop;
exception
    when GLOBAL.BLOCKING =>
--  The evasive action did not solve the problem, so degrade gracefully.
end PHILOSOPHER;
```

Since the Adam compiler does not implement exception propagation during task rendezvous (e.g. rendezvous with the monitor task), evasive action in our experiments uses the value of a parameter, MON_DEAD_FLAG. The evasive action is inserted after the program has been preprocessed since we do not want the evasive action monitor calls to be monitored.

```
task body MONTYPE_PHILOSOPHER is
    MY_ID           : INTEGER;
    MON_DEPEND_ID   : ID_PTR;
    MON_LIST        : ENTRY_PTR;
    MON_DEADLK_FLAG             : BOOLEAN;
    OUTER_DEPENDENTS            : ID_PTR renames MON_DEPEND_ID;
    SEAT            : INTEGER;
begin
    accept SET_ID (N : in INTEGER) do
        MY_ID := N;
    end;
    loop
        delay 1;

        MONITOR.CALLING(MY_ID, TABLE.ID, "SITDO", MON_DEADLK_FLAG);
```

```
        TABLE.TSKOBJ.SITDOWN(MY_ID,SEAT);

        MONITOR.CALLING(MY_ID, FORKS(SEAT).ID,"PICKU",MON_DEADLK_FLAG);
        FORKS (SEAT).TSKOBJ.PICKUP (MY_ID);
        delay 1;

        MONITOR.CALLING(MY_ID, FORKS((SEAT + 1) mod 5).ID, "PICKU",
                                                    MON_DEADLK_FLAG);
        if MON_DEADLK_FLAG then
            MONITOR.TRACE (ALL_TASKS, TRUE);
            MONITOR.UNBLOCK (MY_ID);

            MONITOR.CALLING (MY_ID, FORKS(SEAT).ID,
                                                "PUTDO",MON_DEADLK_FLAG);
            FORKS (SEAT).TSKOBJ.PUTDOWN (MY_ID);

            MONITOR.CALLING (MY_ID, FORKS(SEAT).ID,
                                                "PICKU", MON_DEADLK_FLAG);
            FORKS (SEAT).TSKOBJ.PICKUP (MY_ID);

        end if;

            MONITOR.CALLING(MY_ID, FORKS((SEAT + 1) mod 5).ID, "PICKU",
                                                    MON_DEADLK_FLAG);

        FORKS((SEAT + 1) mod 5).TSKOBJ.PICKUP(MY_ID);
        delay 1;

        MONITOR.CALLING(MY_ID, FORKS(SEAT).ID,"PUTDO", MON_DEADLK_FLAG);
        FORKS (SEAT).TSKOBJ.PUTDOWN (MY_ID);

        MONITOR.CALLING(MY_ID, FORKS((SEAT + 1) mod 5).ID, "PUTDO",
                                                    MON_DEADLK_FLAG);
        FORKS ((SEAT+1) mod 5).TSKOBJ.PUTDOWN (MY_ID);

        MONITOR.CALLING (MY_ID, TABLE.ID, "GETUP", MON_DEADLK_FLAG);
        TABLE.TSKOBJ.GETUP (MY_ID,SEAT);

    end loop;

    MONITOR.END_TASK (MY_ID, MON_DEPEND_ID, MON_DEADLK_FLAG);
end MONTYPE_PHILOSOPHER;
```

7.5 ACTION OF DINING PHILOSOPHERS WITH EVASIVE ACTION

Below is a trace of activity by the evasive version of the dining philosophers. First the monitor description of an imminent dead state is given. A philosopher task is warned, and a trace of its evasive action and subsequent "normal" activity then follows.

Key: See example 7.3.

```
**MON**  GLOBAL DEADLOCK HAS BEEN DETECTED
**MON**    TASK INFORMATION
```

```
0 MAIN  is block_waiting on 11 tasks.
    Its entries are:
            <NONE>
  Its father is: -1

1 TABLE is accepting
    Its entries are:
            SITDO (0*)  GETUP (0*)
  Its father is: 0

2 FORKS is accepting
    Its entries are:
            PUTDO (0*)  PICKU (1)
  Its father is: 0

3 FORKS is accepting
    Its entries are:
            PUTDO (0*)  PICKU (1)
  Its father is: 0

4 FORKS is accepting
    Its entries are:
            PUTDO (0*)  PICKU (1)
  Its father is: 0

5 FORKS is accepting
    Its entries are:
            PUTDO (0*)  PICKU (1)
  Its father is: 0

6 FORKS is accepting
    Its entries are:
            PUTDO (0*)  PICKU (1)
  Its father is: 0

7 SOCRA is calling task number 3 at entry PICKU
    Its entries are:
            <NONE>
  Its father is: 0

8 PLATO is calling task number 4 at entry PICKU
    Its entries are:
            <NONE>
  Its father is: 0

9 ARIST is calling task number 5 at entry PICKU
    Its entries are:
            <NONE>
  Its father is: 0

10 MARX  is calling task number 6 at entry PICKU
    Its entries are:
            <NONE>
  Its father is: 0
```

```
-- RUSSELL will be the philosopher task receiving the monitor warning.

11 RUSSE is calling task number 2 at entry PICKU
    Its entries are:
          <NONE>
  Its father is: 0

**MON**  end of dead state description.

**TRC** call of monitor entry CALLING.

  The consumer is 11 [RUSSE]. The server is 6 [FORKS]. The entry is [PUTDO].

-- This indicates that RUSSELL is taking evasive action and is putting down the lefthand fork
-- instead of attempting to pick up the righthand fork. Note that the monitor call
-- to UNBLOCK, indicating evasive action, is not traced, but must already have been called
-- so that the monitor's "picture" is correct.

**TRC** call of monitor entry START_RENDEZVOUS.
  The consumer is 11 [RUSSE]. The server is 6 [FORKS]. The entry is [PUTDO].

**TRC** call of monitor entry END_RENDEZVOUS.
  The consumer is 11 [RUSSE]. The server is 6 [FORKS].

-- RUSSELL has now put down his left fork.

**TRC** call of monitor entry CALLING.
  The consumer is 11 [RUSSE]. The server is 6 [FORKS]. The entry is [PICKU].

-- RUSSELL now attempts to pickup the lefthand fork again!
-- However he will be behind MARX on the entry queue.

**TRC** call of monitor entry ACCEPTING.
  The server is 6 [FORKS]. The entry is [PICKU].

-- A FORK, task 6, is the only unblocked task.

**TRC** call of monitor entry START_RENDEZVOUS.
  The consumer is 10 [MARX ]. The server is 6 [FORKS]. The entry is [PICKU].

-- Now MARX can pickup his righthand fork, which was RUSSELL's lefthand fork.

**TRC** call of monitor entry END_RENDEZVOUS.
  The consumer is 10 [MARX ]. The server is 6 [FORKS].

**TRC** call of monitor entry ACCEPTING.
  The server is 6 [FORKS]. The entry is [PUTDO].

**TRC** call of monitor entry CALLING.
  The consumer is 10 [MARX ]. The server is 5 [FORKS]. The entry is [PUTDO].

-- Now MARX is finished eating and prepares to put down his forks.

**TRC** call of monitor entry START_RENDEZVOUS.
  The consumer is 10 [MARX ]. The server is 5 [FORKS]. The entry is [PUTDO].
```

```
**TRC** call of monitor entry END_RENDEZVOUS.
  The consumer is 10 [MARX ]. The server is 6 [FORKS].

**TRC** call of monitor entry CALLING.
  The consumer is 10 [MARX ]. The server is 6 [FORKS]. The entry is [PUTDO].

**TRC** call of monitor entry ACCEPTING.
  The server is 6 [FORKS]. The entry is [PICKU].

**TRC** call of monitor entry START_RENDEZVOUS.
  The consumer is 10 [MARX ]. The server is 6 [FORKS]. The entry is [PUTDO].

**TRC** call of monitor entry END_RENDEZVOUS.
  The consumer is 10 [MARX ]. The server is 6 [FORKS].

**TRC** call of monitor entry CALLING.
  The consumer is 10 [MARX ]. The server is 1 [TABLE]. The entry is [GETUP].

**TRC** call of monitor entry ACCEPTING.
  The server is 6 [FORKS]. The entry is [PICKU].

**TRC** call of monitor entry START_RENDEZVOUS.
  The consumer is 9 [ARIST]. The server is 5 [FORKS]. The entry is [PICKU].
```

-- *ARISTOTLE gets his righthand fork and starts eating.*

```
**TRC** call of monitor entry START_RENDEZVOUS.
  The consumer is 10 [MARX ]. The server is 1 [TABLE]. The entry is [GETUP].
```

-- *Now MARX has left the table.*

The trace output continues on indefinitely.

8. REFERENCES

[German 81] German, S.M.
Verifying the Absence of Common Runtime Errors in Computer Programs.
CSD Report Program Verification Group Report PVG-19, STAN-CS-81-866, Stanford University, June, 1981.

[German,Helmbold,Luckham 82]
German, S.M., Helmbold, D.P., and Luckham, D.C.
Monitoring for Deadlocks in Ada Tasking.
In Gargaro, A.B. (editor), *Proceedings of the AdaTec Conference on Ada*, pages 10-25. ACM, Arlington, Virginia, October, 1982.

[Hoare 69] Hoare, C.A.R.
An Axiomatic Basis for Computer Programming.
Communications of the ACM 12(10):576-581, October, 1969.

[Hoare,Wirth 73] Hoare, C.A.R. and Wirth, N.
An Axiomatic Definition of the Programming Language Pascal.
Acta Informatica 2:335-355, 1973.

[Ichbiah et al. 82] Ichbiah, J.D., Krieg-Brueckner, B., Wichmann, B.A., Ledgard, H.F., Heliard, J.-C., Abrial, J.-R., Barnes, J.P.G., Woodger, M., Roubine, O., Hilfinger, P.N., Firth, R.
Reference Manual for the Ada Programming Language: Proposed Standard Document.
US Department of Defense, US Government Printing Office, 1982.
Revised Version.

[Li 82] Li, W.
An Operational Semantics of Multitasking and Exception Handling in Ada.
In Gargaro, A.B. (editor), *Proceedings of the AdaTec Conference on Ada*, pages 138-151. ACM, Arlington, Virginia, October, 1982.

[Luckham et al.,ADAM 81]
Luckham, D.C., Larsen, H.J., Stevenson, D.R., and von Henke, F.
ADAM -- An Ada based Language for Multi-processing.
Program Verification Group Report PVG-20, CSD Report STAN-CS-81-867, Stanford University, July, 1981.

[Luckham,Karp 79]
Luckham, D.C., and Karp, R.A.
Axiomatic Semantics of Concurrent Cyclic Processes.
1979.
Stanford Verification Group report, forthcoming; submitted for publication.

[Pascal Verifier 79] Stanford Verification Group.
Stanford Pascal Verifier User Manual.
CSD Report Program Verification Report PV-11, STAN-CS-79-731, Stanford University, March, 1979.
Editions 1 and 2.

[Taylor 82] Taylor, Richard.
Complexity of Analyzing the Synchronization Structure of Concurrent Programs.
Information and Computer Sciences Report, University of California, Irvine, August, 1982.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.


FIL